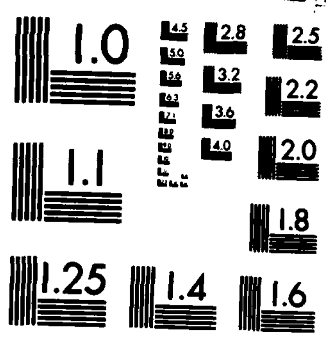
AD-A153 323 THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO A GENERAL PURPOSE..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L R SUZUKI 1/3

UNCLASSIFIED DEC 84 AFIT/GCS/MATH/84D-6 F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(1)

THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO A GENERAL PURPOSE GRAPHICS PACKAGE

THESIS

Laura R. C. Suzuki, B.S.
Second Lieutenant, USAF

AFIT/GCS/MATH/84D-6

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

0

AFIT/GCS/MATH/84D-6

THE ADDITION OF ADVANCED SCENE
RENDERING TECHNIQUES TO A GENERAL
PURPOSE GRAPHICS PACKAGE

THESIS

Laura R. C. Suzuki, B.S.
Second Lieutenant, USAF

AFIT/GCS/MATH/84D-6

Approved for public release; distribution unlimited

AFIT/GCS/MATH/84D-6

THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO A GENERAL PURPOSE GRAPHICS PACKAGE

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Laura R. C. Suzuki, B.S.

Second Lieutenant, USAF

December 1984

Approved for public release; distribution unlimited

Acknowledgements

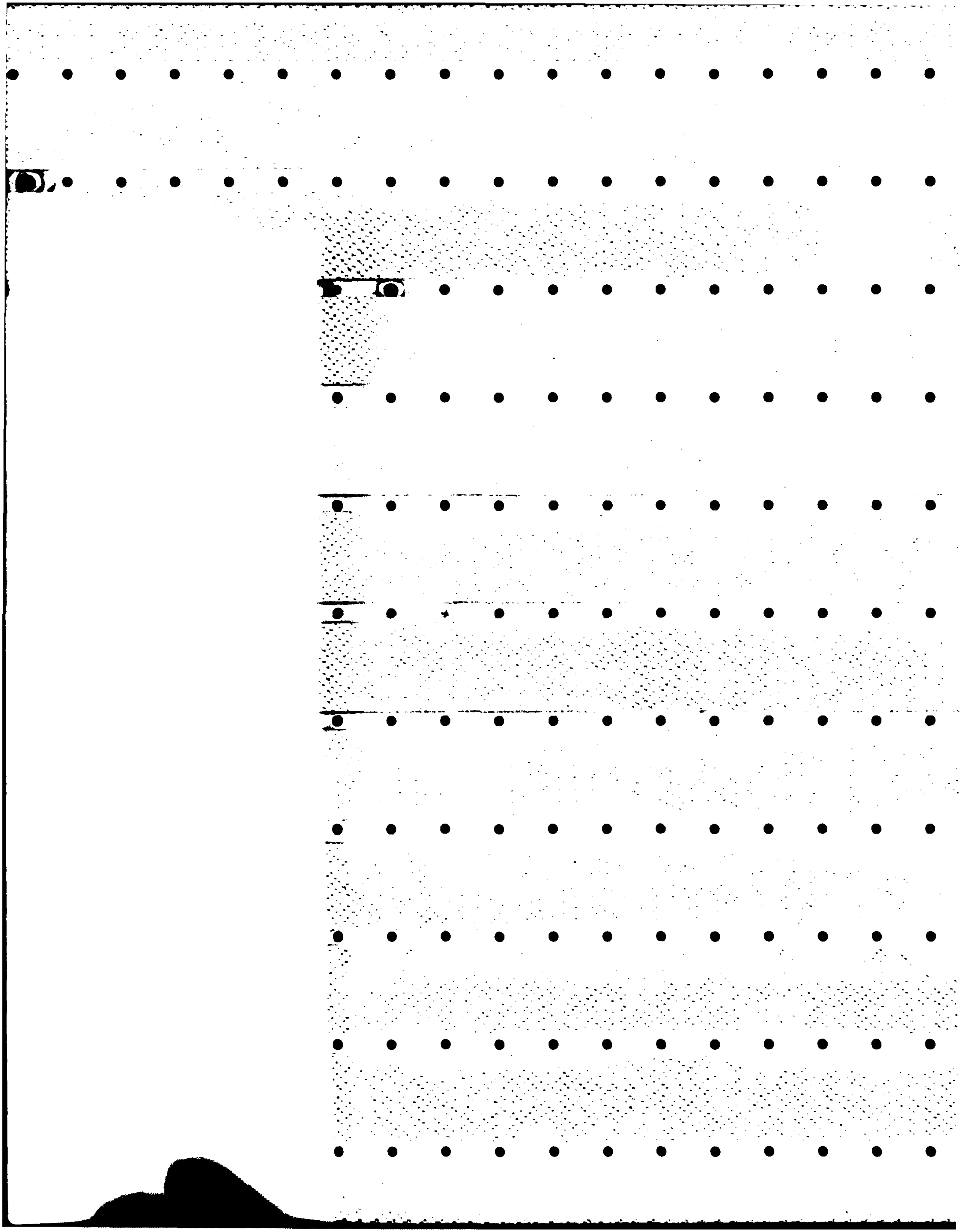
Many people have helped me on this thesis effort. Of course, there is my thesis advisor, Prof. Charles W. Richard, along with a few professors in the Physics Department, especially Maj. Wharton, the "optics guy." Several of the professors in the Mathematics Department were helpful when I was looking for shortcuts and better methods in my equations. And a special thanks to the men at the modeling shop, who fabricated a couple of "real" objects so that I could test my pictures against an actual scene.

However, the person who deserves the most thanks is my husband, Yoshiaki Suzuki. He typed, word processed, checked spellings, and checked grammer on my thesis. He also comforted me when things looked bleak, and put up with numerous late night programming sessions running on into late morning, and sometimes late afternoon. Without him, I doubt if I would have had the strength to complete this thesis.

Table of Contents

List of Figures

List of Tables



List of Tables

Abstract

In this thesis

A method for generating realistic scenes by computer graphics was investigated. The algorithm which was used was a ray tracing algorithm. The scenes it was capable of rendering included those containing transparent and reflective surfaces. The implemented surface types were planar polygons and spheres. Minor surface irregularities were simulated for specular reflection from light sources. The resulting package was added to an implementation of a CORE graphics system (written in Pascal), to serve as its hidden surface removal facility. Additional keywords: CORE [illegible] program; requirements; test and evaluation; pictures; photographs; light sources; ambient light; reflectivity; color resolution.

THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO A GENERAL PURPOSE GRAPHICS PACKAGE

I. Introduction

Background

Much research into realism in computer graphics is under way. The point has been reached where many are no longer satisfied with wire frame houses and polygonally surfaced cars on their graphics screen; they want bold shadows behind their skyscrapers, delicate shading on their porcelain vases, and one-way glass on their limousines. In other words, people are tired of their graphics looking like the Saturday morning cartoons, they want Rembrandts.

A simple wire frame drawing of a house would be recognized by most people, even though it does not look much like a house. The same house drawn out of appropriately colored polygons begins to look like a house. Adding detail like shingles on the roof, a lawn in front, and landscaping makes the picture look at least as good as what is seen in cartoons. But when one adds the shadows from the trees, the sunlight's reflections from the clear glass windows, the shading on the warped siding, and natural textures, then it begins to look real. In a nutshell, realism in graphics is

that which makes graphical output resemble reality.

There are features of natural scenes that are not usually included in general purpose graphics packages. Examples of transparency, reflection, texture, shadows, and shading can be seen in everyday life, but are not usually simulated by computer. While this effort did not add all of these features to a graphics package, it did add some, and to an extent, increased the capability to produce realistic graphics at the Air Force Institute of Technology (AFIT).

The method used to do this was a ray tracing algorithm. In ray tracing, simulated light rays are traced backward from their ending point (the eye point), through the logical reflections and refractions that they must have gone through, to estimate the color and intensity of light that reaches the eye from a given direction. It takes a good deal of time, compared with other algorithms for adding features like transparency, but it can produce extremely good results, and can be extended to provide aspects of realism that have not been included in this effort, such as natural textures.

Ray tracing algorithms are good at rendering scenes, but are almost useless without a way to define the scenes. However, such an algorithm could be a useful addition to an established graphics package. A general purpose graphics package can lend its scene generation ability to the ray tracing algorithm, and gain the capability to produce realistic scenes.

On AFIT's Vax 11/780 under UNIX, there is an implementation of CORE (21), a general purpose graphics package written in Pascal, from the University of Pennsylvania (UPCORE). When this CORE implementaion was rehosted to run in its present environment at AFIT by John W. Taylor (22), the only color device to be interfaced with it was a color plotter, which is unsuitable for the kind of picture that ray tracing produces. However, other color devices now exist at AFIT, and in the course of this effort, were interfaced with the package.

To summarize what has been done in this effort, a ray tracing algorithm has been implemented. That implementation has been interfaced with an existing graphics package, the University of Pennsylvania implementation of the CORE package (UPCORE). Device drivers for two new color devices were added to UPCORE, to be able to see the results of the ray tracing implementation clearly.

Literature Review

An initial literature review was done in the very general and rather wide area of adding realism to graphics. The purpose was to find out what techniques existed, and the emphasized subject areas were reflections, texture, transparencies, and shading. The results of this review are in the subsection "Research in Realism" of this chapter.

After a preliminary review of the literature on realism, it was decided that a ray tracing algorithm would

be used. Ray tracing has more of a basis in optics than other scene rendering methods. Many desirable features, such as shading and transparency, can be combined into one algorithm by simply applying the appropriate optics principles to the problem. A summary of current research is included in the subsection "Research in Ray Tracing" of this chapter.

Research in Realism. Dungan and others did research into texture tiles in 1978 (9). The reason for their research was to find ways to add apparent texture to otherwise untextured surfaces, to enhance the depth and relative location of objects in a scene. The main problems encountered were aliasing (insufficient resolution to show fine patterns correctly) and macro patterning (undesirable visible patterns formed by large numbers of duplicate tiles). The results achieved were suitable for the purpose they were intented for, but did not present a very realistic impression of texture.

In 1974, Blinn and Newell developed a technique to map patterns and pictures onto a wider variety of shapes (4). The technique used the same methods as are used to create bivariate surface patches. The picture or pattern is distorted and then mapped onto the patches. This technique gives the impression of a glazed surface, with a picture or pattern painted under the glaze. Since shading is done on the outside of the object, regardless of the picture mapped upon it, the object does not look truely textured.

In 1978, Blinn did significant work in creating objects with a true appearance of texture (3). His approach is to add a function defining the texture to the function defining the surface. Then the hidden surface, shading, and reflection algorithms are carried out on the new surface. The results of this technique are impressive. Not only do the patterns show appropriate shading, but the silhouette view also shows evidences of the texture. Any functionally determined pattern can be used. There are still problems with this technique in that aliasing occurs when a texture is mapped into a small area. However, anti-aliasing techniques can be used to combat this.

Transparency was discussed by Kay and Greenberg in 1978 (16). They extended the traditional approach of varying the intensity of light depending upon the amount of material light is traveling through, to actually simulating refraction of light as it travels through a different medium. This technique is fairly good, especially in complex scenes where ray tracing would take far too much time.

Simulated reflection in computer graphics often gives an object a plastic appearance. Cook and Torrance investigated this problem and found that it is because most reflection algorithms give an object diffuse reflection of the color of the object, but have the object reflect specularly all colors of light equally (7). This is characteristic of plastic. Most materials reflect one color

of light more strongly than other light colors, so that the specularly reflected light is no longer white.

Research in Ray Tracing. The first graphics involving a simulation of the process light goes through to reach a viewer was by Appel (1). His method was to bombard a scene with simulated light rays, and count the number that impinged upon an area to determine the intensity of the light in that area. However, the results obtained for randomly produced light rays were poor. For systematically produced light, the results were better, but time consuming.

Goldstein and Nagel developed a more advanced version of ray tracing, which was a simulation of a camera and a light source (12). In this method, instead of tracing the light rays from the light source to the camera, the light rays are traced in the reversed order, from the camera to the light source. The intensity of light at a point is given as (12:27)

$$I = I_0 \, k \cos \theta + A_0 \tag{1-1}$$

where I_0 is the intensity of the light source, k is the coefficient of reflection of the surface, θ is the angle between the normal to the surface and the direction of the light source. This scheme only allows for diffuse reflection, giving all surfaces a chalk-like appearance. However, the concept of having transparent objects was introduced here. Despite the limitations of this simple

representation, some well-shaded pictures were produced.

In 1980, Whitted described a ray tracing algorithm that adapted Phong's light model (19) to ray tracing techniques to give a good approximation of specular reflection. Phong's model as written in Whitted's notation is expressed as

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j) + k_s \sum_{j=1}^{j=ls} (N \cdot L'_j)^n \qquad (1-2)$$

where I is the reflected intensity, I_a is the ambient light, k_d is the diffuse reflection constant, $\bar{N}$ is the unit surface normal vector, L_j is the vector in the direction of the j-th light source, k_s is the spectular reflection coefficient, L'_j is the vector in the direction half-way between the viewer and the j-th light source, and n is a constant that determines the glossiness of the surface. Phong's model is acceptable for very simple scenes, but in cases where the object is smooth and very shiny, the model does not show the specular reflection accurately.

Whitted adds light from transmission (through transparent surfaces) to the model, and ray traces to find the intensity of the light in the specular reflection and transmission terms. His new model is

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j) + k_s S + k_t T \qquad (1-3)$$

where S is the intensity of the light along the incident

angle, k_t is the transmission coefficient, T is the intensity of the refracted light, and all other symbols are as defined before. This new model will handle reflection and refraction, but does not produce realistic specular reflections on less glossy objects as well as Phong's model. The reason for this is that less glossy objects have micro-facets that vary slightly from the surface norm. These facets reflect a good deal of light seen in a specular reflection, but are not accounted for in Whitted's model. Whitted overcomes this by ray tracing multiple rays for each reflection and averaging the results with each ray perturbed by an amount dependent upon the glossiness of the surface. When the specular reflection is caused by a light source, Phong's model is used.

Kajiya, in 1982 and 1983, discussed the problems and solutions of ray tracing several kinds of non-planar surfaces (15). Since the majority of the time spent in a ray tracing routine is spent finding the intersections of rays and surfaces, Kajiya goes into faster methods for solving the intersections of rays and non-planar surfaces. He discusses solutions for ray intersections with parametric patches and presents an algorithm that can solve the problem using straight line code. This sequential approach means that a pipeline could be built to process these calculations at a faster speed.

Kajiya presents an algorithm to ray trace fractal surfaces without generating the entire fractal surface at

once (14). Instead, he encloses an area of the surface with a volume that has a high probability of containing the entire area of the generated surface. Only if a ray intersects this volume does the surface need to be generated. Kajiya also talks about methods to intersect rays with prisms and surfaces of revolution.

Hanrahan, in 1983, discussed methods to intersect algebraic surfaces, that is, surfaces that are a function of a polynomial (13). Hanrahan chooses to substitute the equation for the ray into the equation of the surface, then solves the resulting equation. There are general methods for solving equations up to order 5. Functions of a greater order can be solved by using an iterative algorithm that gives an answer exact to with a given tolerance.

Problem Statement

AFIT has a version of CORE (21) produced by the University of Pennsylvania (UPCORE) and rehosted to the AFIT Vax 11/780 under UNIX operating system by John Taylor (22). CORE is a well defined package (21), but it has some severe limitations. For example, it has no light models, shading, or shadow generation. Also, the University of Pennsylvania implementation does not follow the CORE specifications completely or exactly. However, it is one of the few packages currently available for student work.

It would be advantageous to be able to use the CORE graphics package to define objects, and then to be able to

produce a realistic rendering of the scene in question by use of an advanced algorithm. This would give the user all of the advantages of working with a well defined package, and also the advantages of an advanced scene rendering technique.

Ray tracing is a good scene rendering algorithm, even if it is a rather slow algorithm. While it could be used with a package such as CORE, CORE only contains planar surfaces, which do not exercise the algorithm thoroughly, and hence are not extremely useful to test the implementation.

Because CORE can not test the ray tracing implementation thoroughly, it is necessary to do some testing outside of CORE, using such surfaces as spheres which are more diagnostic of logic flaws.

In summary, advanced scene rendering capabilities are to be added to an implementation of CORE. The scene rendering algorithm chosen for implementation was a ray tracing algorithm. Also, the color graphics capabilities of the CORE package are to be further enhanced by the addition of new color devices to the package.

Overview of Thesis

The next chapter discusses the requirements and design of the software developed in this effort. This does not include the design of the ray tracing implementation, which is instead discussed in Chapter 3. The design of the

interface between the ray tracing package and CORE, and the design of the device drivers written for inclusion with the CORE package are also included in Chapter 3. The testing and evaluation results are in the fourth chapter. The fifth chapter contains results and conclusions.

II. Requirements and Design

The major thrust of this effort was implementing a ray tracing algorithm. Requirements and general design are discussed in this chapter and the detailed design of the ray tracing implementation is given in the following chapter.

This chapter begins with the requirements for an advanced scene rendering algorithm to be added to CORE. The choice of the ray tracing algorithm is discussed, and a description of a ray tracing algorithm and some of its approximations is given. The detailed requirements for adding this particular implementation of the algorithm to UPCORE are given next. The requirements for the device drivers are then discussed.

The design portion of this chapter consists of the design of the interfaces and the design of the device drivers. The interfaces include the interface between UPCORE and the ray tracing package, the interface between UPCORE and the new device drivers, and the interface between the ray tracing package and the device drivers. The driver design includes a general design plus a detailed design for each of the devices that drivers were to be written for.

Requirements For the Addition of Advanced Screen Generating Techniques to a CORE Package

The techniques used to produce the desired effects

(i.e., shading, transparency, hidden surface removal, reflections) must be compatible with each other. For example, if shading is to be done, it must not prevent the transparency algorithm from working on the shaded scene. The usefulness of adding these techniques is greatly reduced if they cannot work together to combine their benefits.

Since CORE does not define some of the objects and attributes needed for these advanced techniques, such as light sources and transparency, it will be necessary to develop definitions for such items. This is necessary because, without certain additions, the effort of adding advanced capabilities to CORE would be worthless. All of the new user definable attributes and objects must be invoked with code which is compatible in format and style with that which is already defined in the CORE standard. This is so that a user familiar with CORE will be able to use the new additions to CORE without experiencing discomfort about their usage.

The addition of the algorithms must cause as little structural change as possible in the implementation of CORE used. Making major changes is more likely to introduce new bugs to the system. Also, techniques that require major changes to the UPCORE implementation are likely to require major changes of other CORE implementations as well, reducing the overall benefits of adding advanced scene rendering techniques to CORE in general.

Choice of Algorithm

There are several hidden surface, transparency, and shading algorithms. However, most algorithms for these three different problems work independently of each other, and place limitations on the shapes or coloring of objects (1, 2, 4, 7, 11, 13, 14, 15, 16, 17, 19, 25). Ray tracing algorithms do not have these disadvantages. Ray tracing algorithms can allow any shape for which the intersections with a simulated light ray can be found. There are no restrictions on color. They can be expanded to any resolution. Transparency is allowed and can be handled correctly. In addition, hidden surface removal, shading, and shadow production can be done as one process. For these reasons, it was decided to implement a ray tracing algorithm.

An adaption of Whitted's illumination model (25) was chosen to use in this implementation. While Whitted's model has been used on a Vax 11/780 under UNIX, which would lead one to believe that it is not too complex to be implemented on AFIT's Vax 11/780, also under UNIX, this proved not to be the case. This implementation has been done in Pascal, not the best language for UNIX, while Whitted's implementation was done in C, ideal for UNIX. Also, AFIT's Vax 11/780 is very heavily used, so that it became difficult to complete computationally bound jobs in a reasonable amount of time. For this reason, Whitted's model was simplified somewhat to reduce computation time.

Introduction to Ray Tracing

Ray tracing for computer graphics has its foundation in optics. Because of this, with unlimited computer resources, it is possible to produce an almost perfect representation of a scene, perfect to the extent that classical optics can make it. However, without unlimited computer resources, it becomes necessary to make simplifying assumptions about the behavior of light.

The conventional method of ray tracing for computer graphics is to start at the eye point of a picture, and to trace a simulated light ray in reverse, through each point of the view plane corresponding to pixels in the window of the output device, to find out where it must have come from, and hence, what it must be composed of, to have reached the eye point. The light rays are reflected, refracted, and split where appropriate, in the attempt to find the light source of their origin. In theory, this process continues until the light source origin of each ray is found (see Figure II-1). But considering the rate at which simulated light rays would have to be generated to get an estimate of where the light is deriving from, many necessary approximations must be made on the number and directions of light rays to be traced.

Approximation 1. It is not necessary to trace rays all of the way back to a light source. Instead, there are equations (see, e.g., (25)) to approximate the amount of

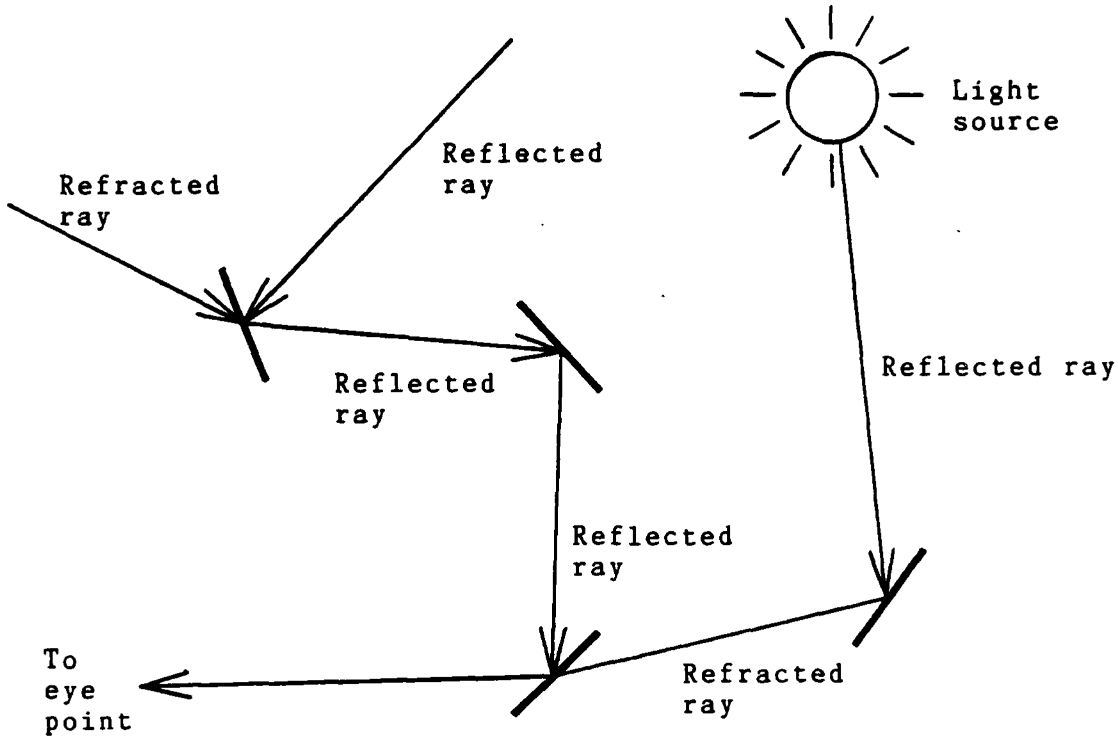


Figure II-1. Simplified Ray Tracing

light reflected along a given ray from a point on a surface lit by a particular light source (see Figure II-2). So, the contribution from light sources is calculated at each point of a surface that is intercepted, rather than trying to trace every ray back to the light source.

Often, a transparent object will be between a point and a light source. Light coming through that object will most likely be refracted, so that it is not striking the point at the same angle, intensity, or color as it would be if the object were not there. Part of this is because, while refracting through the object, the light will be bent, so that the path that a light ray takes between the light

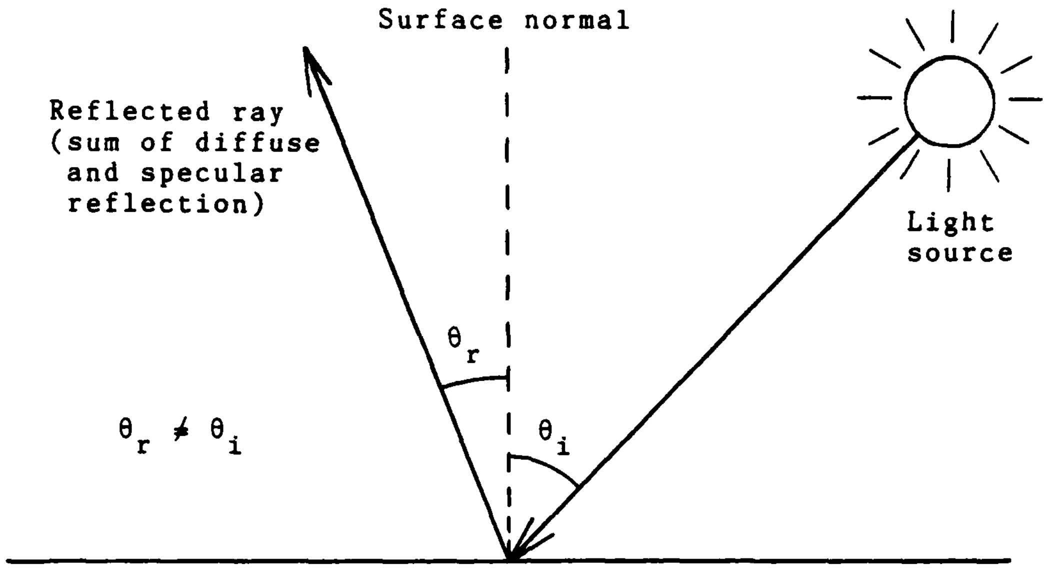


Figure II-2. Reflected Light From a Light Source

source and the point is not a straight line. The problem of finding the correct path, possibly through several different objects with different refractivities, when a path may or may not even exist, is a difficult one.

Approximation 2. The path that light travels when going through transparent objects can be approximated by a straight line. The intensity and color of light traveling along that line can be determined by the transparency and color of the objects it is traveling through. This approximation will quite often produce totally erroneous results. However, it produces a logically appearing, although incorrect, picture. The shadows from semi-transparent objects appear the color of the object, and the thicker the object, the darker the shadow.

Once the illumination from light sources at a point on a surface is accounted for, that leaves the indirect illumination from light being reflected from other objects. This indirect illumination is caused by a large number of rays, far too many to trace individually.

Approximation 3. Only a small sampling of light rays reflected from other surfaces needs to be traced. This approximation is justified by the fact that in most cases the majority of the illumination is directly from light sources.

The sample of rays that is appropriate is dependent on the surface type. A shiny surface, such as a mirror, should result in most light rays being reflected with an angle of reflection nearly equal to the angle of incidence. On the other hand, the appearance of a dull surface is affected evenly by light from all directions.

After the contribution from reflected light is found, there is yet the refracted light to be accounted for. Refracted light is that light which is traveling through a surface instead of reflecting off of it. As light passed through a surface, it is bent by an amount dependent upon the index of refractivity of the media on either side of a surface and the angle at which the ray strikes the surface.

The amount of both refracted and relected light is dependent upon the indices of refractivity of the media on either side of a surface as well as the angle at which the light strikes the surface. An object with a higher

refractivity will reflect more and refract less light than an object with a lower refractivity, if both objects are in a medium of lower refractivity than themselves.

Much light is contributed to a scene from light being multiply reflected off of and refracted through many surfaces. This light causes low level illumination in places in which no light sources are directly illuminating.

Approximation 4. Low level background light can be approximated by light of a constant amount and color, radiating in all directions. This approximation greatly reduces the number of reflected and refracted rays that must be traced to produce a realistic representation of shadows and indirect lighting.

The color of light which is reflected off of a surface depends upon the color of the incident light and certain properties of the surface, including color. Light diffusely reflected off of a surface can only be light containing colors present in that surface. However, it also can only contain colors present in the incident light. For specular reflection, however, most surfaces reflect light of approximately the same color as the incident light, regardless of the color of the surface. Some shiny surfaces, like gold, specularly reflect light of one color more than others, but the color change is slight.

Approximation 5. The specular reflections off of any surface can be assumed to be the same color as the incident light. This is justified by the fact that to simulate

surfaces that reflect one color more strongly than the others, or surfaces that reflect different colors more strongly at different angles, would add greatly to the computational time needed to produce a picture.

Absolutely clear objects (i.e., white, transparent objects) transmit all colors of light. However, not all objects are clear, so that some transparent objects transmit some colors of light more strongly than others. An object can only transmit a color of light that is both in the color of the incident light and in the color of the object.

While ideally, one should trace numerous rays for each pixel (smallest area of resolution) of the output device, this is often not necessary or possible due to time constraints.

Approximation 6. It is often possible to calculate the color of large areas to be displayed on the output device by approximating the color of the area by the average of the colors at the four corners of the area. This minimum area resolution, or largest area that can be calculated in this way, can greatly reduce the number of rays that need to be traced.

Specifying a minimal are resolution can greatly reduce the number of rays which need to be traced, but unless there are some criteria to subdivide this area when needed, detail of the scene can be lost due to insufficient resolution. One criterion would be when the color of the four corners of an area are different. However, a very small difference in

color may not be visible to human eyes, so to subdivide on the basis of any difference in color would be a waste of time.

Approximation 7. An area only needs to be subdivided if the colors present at each corner of the area are different by more than a certain amount. That amount, the minimum color resolution, is the maximum difference between the components of two colors that is acceptable if the two colors are to be considered the same.

Requirements For a Ray Tracing Package

The ray tracing package must be as independent of UPCORE as possible. This is especially important because of the incompleteness of the University of Pennsylvania CORE package presently on the AFIT Vax11/780 UNIX system. Since this package is not a full implementation of CORE, there exists the strong possibility that a better package will be obtained or developed that will be of more use. Since, the UPCORE package requires a good deal of system resources, it probably will not be maintained if a better package becomes available. So, to facilitate future usefulness of this effort, the use of CORE modules and data structures by ray trace package modules must be limited to a minimum number of interface modules, whose main function will be the transfer of data.

Special emphasis must be placed on the documentation of these modules. This documentation must explicitly state the

purpose and format of the data transferred by modules in order to facilitate the writing of new interface modules for any new graphics package that may be placed on the system.

CORE graphics lacks many of the attributes that are necessary for realistic picture generation. The ray trace package must be able to support these attributes. This implies that modules to set these attributes must be designed, default values must be specified, and additional data structures must be developed and added to the CORE package. The procedures must be similar in parameters and appearance to the attribute setting modules that exist in CORE.

CORE can only create objects from polygons. One of the advantages of ray tracing algorithms is that they do not require a surface to be approximated by polygons. Because of this, the ray tracing package must not rely on all surfaces being planar, for future uses may be with a graphics package containing curved surface representation. Instead, the requirements for an interface with a package allowing other surfaces must be thoroughly documented.

Multiple light sources should be allowed. The user should be able to specify parallel, point, and ambient light sources. He should also be able to specify the color and intensity of the light for each source. This requires that data structures to handle these new objects be developed and added to CORE.

The user must be able to choose between parallel and

perspective projection. This choice should be indicated by use of the appropriate CORE module.

The user should be able to specify a maximum depth of ray tree. In this way, the user can simplify the use of the ray tracing package for simple scenes where repeated reflections or refractions do not play an important part in the final picture.

The user should be able to specify the minimal area resolution of the picture. This gives the user the opportunity to trade accuracy for time in cases where time is at a premium.

The user should be able to specify the minimal color resolution of an area. This permits a user to be able to get a quick view of a scene consisting of many color changes by sacrificing the realism in the picture.

Requirements For Device Drivers Being Developed For UPCORE

The invocation of the device drivers should resemble closely those of the invocation of the already existing drivers in use with UPCORE. Any changes should not require significant structural changes of CORE. This is because the purpose of device driver packages is to free programs from the idiosyncrasies of the individual devices.

However, it is often desirable for a graphics package to be able to use special features that a device may support in hardware, to prevent having to do the same operation in software. So, the device drivers must make useful hardware

features available to UPCORE.

CORE - Ray Trace Interface

Ray tracing can be thought of as primarily a hidden surface removal algorithm. CORE supports hidden surface removal, but UPCORE has no hidden surface removal facilities, although the procedures specified to invoke hidden surface removal had been implemented in partial fulfillment of the requirements of the CORE standards. Therefore, the invocation of the ray tracing package was done in the manner specified by CORE to invoke hidden surface removal. That is, the display mode is set to remove hidden surfaces, and a batching of updates is invoked (see Figure II-3).

It was necessary to make certain changes to UPCORE so that it could suport this ray tracing package. UPCORE does clipping of primitives except for the case where clipping is turned off. Because clipping is a desirable feature of CORE, and because ray tracing requires unclipped coordinates, it was decided to save the original coordinates of the primitives for use in ray tracing, rather than use the coordinates already stored which would require that clipping be turned off. This required that additional data structures be added to UPCORE to store the original coordinates. Basically, the data structures already in existence for storing coordinates were duplicated to store original, unclipped, coordinates.

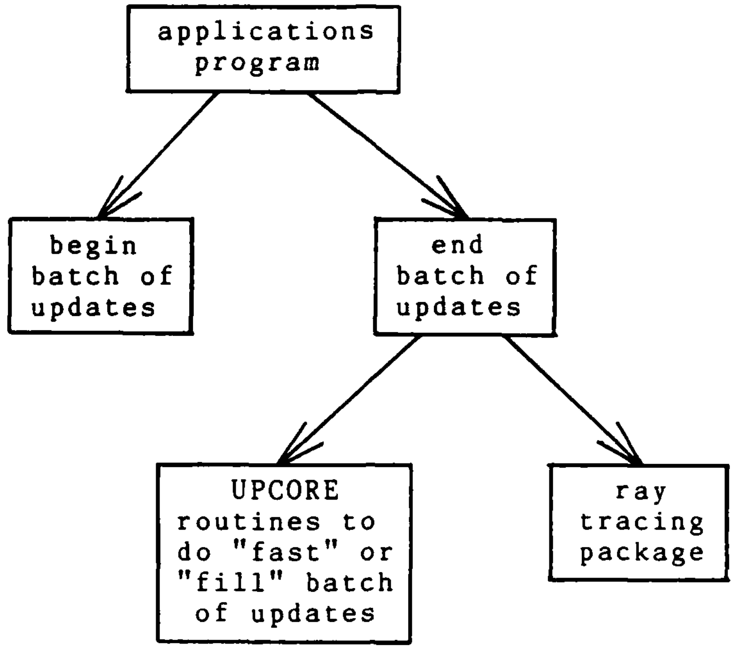


Figure II-3. UPCORE - Ray Tracing Interface

The ray tracing package works in a coordinate system where the eye point (for a perspective projection) is at the origin and the view plane (for a parallel or prospective projection) is perpendicular to the z-axis along the positive z-axis. Because UPCORE calculates a transformation matrix to do this, the UPCORE matrix is used instead of a new matrix being calculated. Also, the user definable world coordinate transformation matrix is used. However, segment transformations are not allowed. A three-dimensional viewing pipeline is shown in Figure II-4, and the ray tracing interface with the CORE viewing pipeline is shown in Figure II-5.

Also, UPCORE does not have polygon clipping. It only

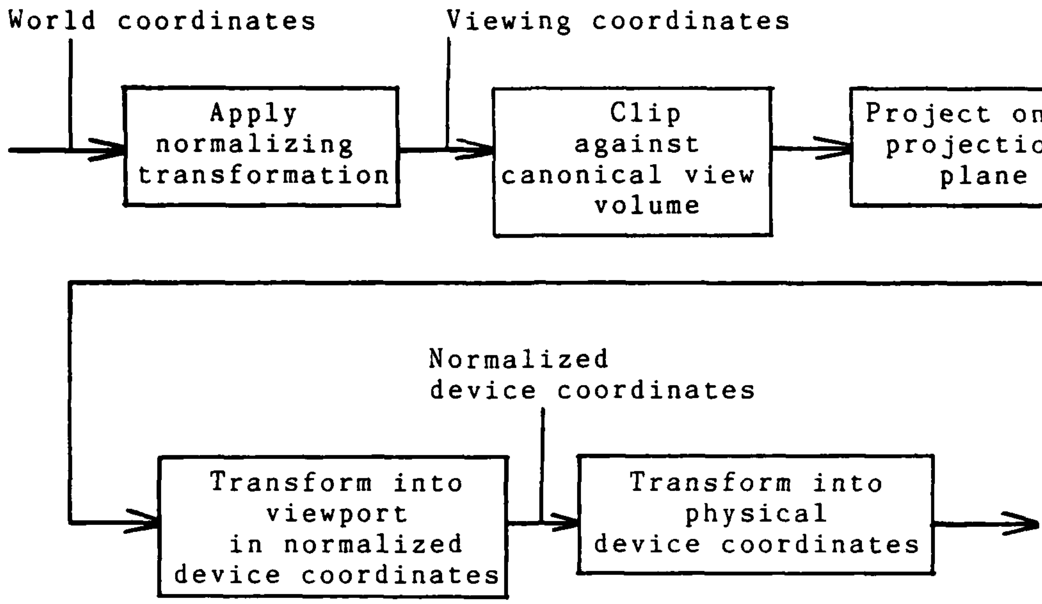


Figure II-4. Implementation of 3D Viewing (10:284)

stores the clipped edges of the polygon, and a scan line algorithm is used to output a filled polygon one scan time at a time. In addition to producing a good deal of output, this method does not always produce correct fills for clipped coordinates. Because of the considerable amount of output generated by the package, it was considered best to write device drivers to take advantage of the special hardware features which output filled areas. Since these features were of no use to CORE, as it stood, and considerable modifications to UPCORE would be necessary to add polygon clipping to UPCORE, it was considered simpler to add the device polygon fill features to the ray tracing package and have the ray tracing package do its own I/O

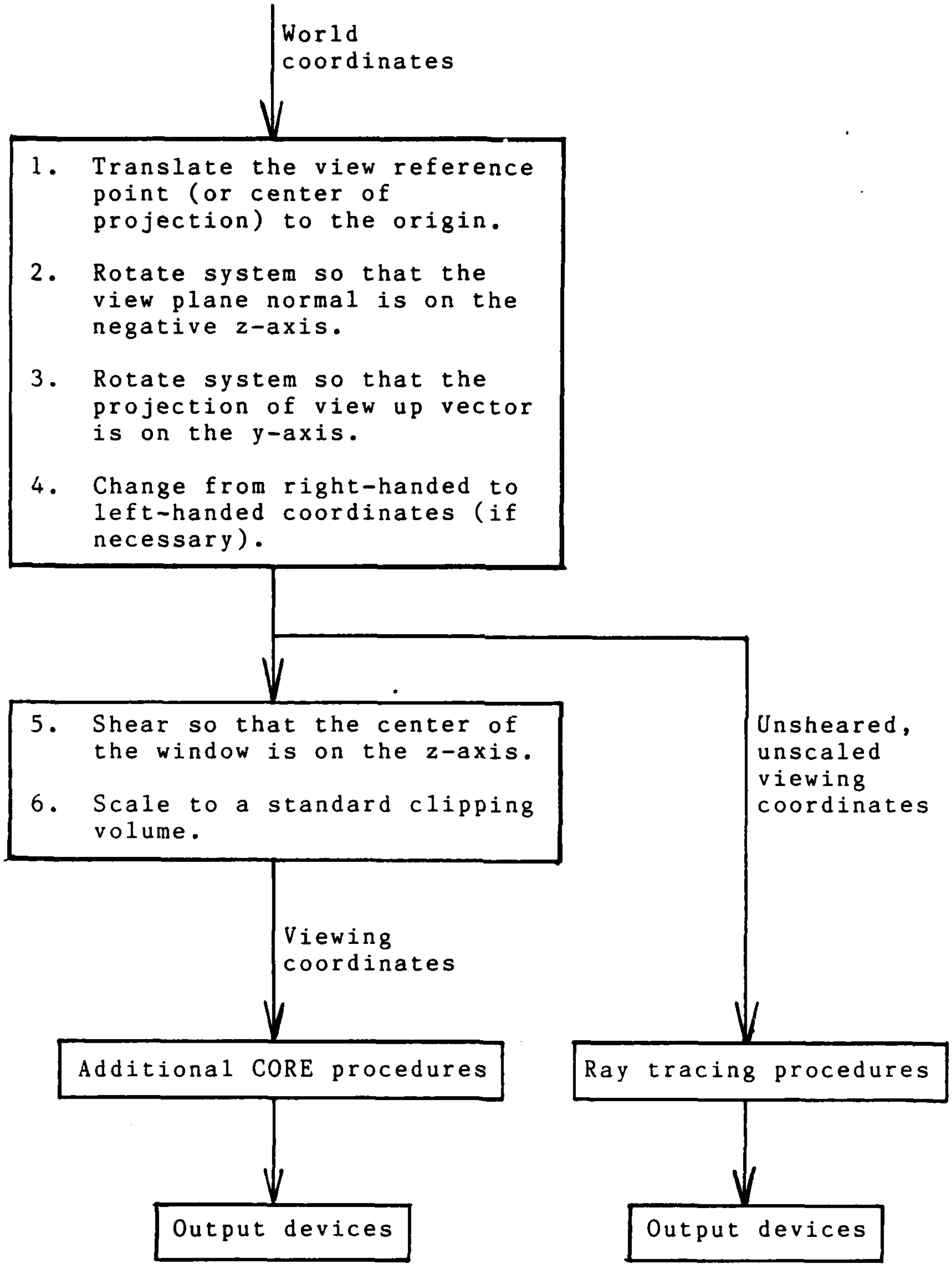


Figure II-5. Ray Tracing Interface With UPCORE 3D Viewing Pipeline

rather than referring it back to CORE, even though this was not good programming practice. However, the I/O procedure in the ray tracing package "pddrawrec," is structured in such a way that it could easily be changed at a future time to make calls back to UPCORE or some other graphics package to access its I/O procedures.

CORE - Driver Interface

The CORE - driver interfaces were fairly well determined by the existing CORE - driver interfaces for other output devices. In most cases, the only modifications that were needed were to add additional conditional constructs similar to those in existence for other devices. In only one case was it necessary to write original code, which was to choose the color for primitives to be drawn in on the output device. Even there, it was only necessary to re-code an existing procedure that was stubbed out when UPCORE was rehosted to the AFIT Vax 11/780 under UNIX. So, in practicality, no major design decisions had to be made pertaining to the CORE drivers interfaces.

Ray Tracing - Driver Interface

The ray tracing package has a very restricted set of output needs. Because of this, all of the I/O is actually done by one procedure, "pddrawrec." Currently, "pddrawrec" calls the device drivers itself. It would be better to have "pddrawrec" call pertinent procedures in UPCORE to do the

I/O. However, no appropriate procedures exist in UPCORE.

The necessary procedures for both output devices are ones to output rectangles. To minimize the length of the output files, "pddrawrec" also calls routines for drawing lines and point, for cases where a rectangle has degenerated to a line or a point, for the Model One/25S device, where the difference in length of the command is significant.

Driver Design

<u>General Design</u>. The functions that were chosen to be implemented in the driver package were chosen basically to be functions required by this UPCORE implementation. In addition, routines that could help to reduce the size of output files produced by the ray tracing package were also done. The necessary routines for UPCORE were determined by examining the driver routines that were already integrated with UPCORE, plus determining by inspection places in this CORE implementation that dealt with color. Most of code in UPCORE which dealt with color had been rendered inoperable when this implementation was rehosted to run on AFIT's Vax 11/780 under UNIX, because at that time, there were no color devices available other than a plotter. The only additional routines, other than those equivalent to existing routines, were the routines to set the color that primitives are to be drawn in.

The ray tracing package could have been designed to use only those drivers that needed to be implemented for CORE.

However, it is much more convenient to be able to draw filled rectangles, which CORE was only capable of doing by outputing one scan line at a time to fill the area. So, to better support the ray tracing package, routines to output rectangles were written. Detailed descriptions of both sets of drivers are in Appendix D.

Tektronix 4027. The Tektronix 4027 is only capable of displaying eight colors on its screen at once. However, the ray tracing package requires a much wider range of color in order to give acceptable results. So, the design of the device drivers for Tektronix 4027 revolved around the simulation of shades of colors by means of mixing dots of the eight colors to produce an average of the desired shade. Fortunately, methods exist to do this, and an implementation of such a method intended for the Tektronix 4027 was available for adaptation (18).

This method works fastest if some data structures can be initialized to values just once before the rest of the package is used. It became necessary to require that an initializing routine be called before the package was used. Because CORE was slightly inconsistent in the order in which devices were initialized and used, it became necessary to have several of the routines in the driver package call the initializing routine if that routine had not been called previously. This is not a good form for a package of this type, but it was deemed the best solution at the time.

Several internal routines were required to deal with

the generation of patterns to simulate shades of color. These routines were required to do such things as set the current color to a particular color or pattern, as well as to define new patterns as needed.

The Tektronix 4027 has an annoying peculiarity that was only partially handled by these drivers. The peculiarity involves the way in which the device handles the patterns which needed to be set in order to generate the necessary shades of colors. The Tektronix 4027 terminal has memory for 120 user definable patterns. Occasionally, one of these patterns needs to be redefined to accomodate a new shade. Any character block filled with a pattern which is not a part of a border of a color, and has had any other changes made to it since the initial fill was done, will change to a new pattern if that pattern number is redefined. This will not affect the filled polygons output by UPCORE, since they are filled one line at a time and not as a block. Also, the routine to output a rectangle for the ray tracing package was written to draw lines across the rectangle in sufficiently many places so that the pattern would be fixed in place. However, the clear screen routine does nothing to alleviate this, so that if a background color is set to an off-shade in UPCORE, large areas of the screen may abruptly change color if the pattern of the background color is redefined.

Model One/25S. The Raster Technologies Model One/25S device is a highly sophisticated device with numerous

options and operations supported. It also can display around 65,000,000 colors, although not all at once, since the screen is only 512 by 512 (= 262,144 pixels). This eliminated many of the problems encountered with the Tektronix 4027 terminal. Only one special routine needed to be written to support the other routines, and that was a routine to send hexadecimal values to the terminal.

The main problem with the Model One/25S terminal was that the default is for it to accept 8 bit binary graphics commands from the host computer. However, AFIT's Vax 11/780 ordinarily sends 7 bit characters, and does not seem to send certain control characters. Because of this, certain changes had to be made to the special character set of the device to be able to use it. The special characters are stored in non-volatile memory, so except for adverse conditions, do not need to be reset. A listing of the commands necessary to set up the terminal in case of a memory failure are listed in Appendix D.

The Model One/25S terminal is a little unusual in that it is not possible to simply reset all of the coordinate registers. This means that to reset the device coordinate system from one set of values to some desired set of values, it is first necessary to find out what the current coordinate values are. So, the driver must request input from the device in order to reset coordinate system. This means that if the program is to run to an output file, instead of to the screen, it is necessary to include the

default values of the coordinate system registers as input and then give the terminal a cold boot before directing the output to the screen. This is discussed in Appendix D.

Because the Model One/25S is a very versatile device, many of the commands that needed to be issued to the terminal for CORE to be able to use it were coded as separate routines, despite the fact that neither CORE nor the ray tracing package called them directly. This was done so that the device driver package would have more general usefulness. Also, it is possible to add new routines to the package to utilize more of the device without extensive reworking of the original package.

III. Detailed Design of Ray Tracing Package

The major thrust of this thesis was the implementation of a ray tracing package to be added to UPCORE. Correspondingly, the major portion of this thesis is about the details of the design of the ray tracing package.

Optics plays a vital part of ray tracing. So to start this chapter, a brief discussion of the useful equations from optics is given. A detailed discussion of the light sources, additional surface attributes, and other parameters that were added to CORE is given. Light sources that are discussed are point, parallel, and ambient sources. The attributes that are discussed are reflectivity, refractivity, transparency, and smoothness. The other useful parameters that were added are ray tree depth, area resolution, color resolution, light scaling, and distance scaling.

Discussions are also given about the surface types which have been supported in the ray tracing package, and the reason for them.

Picture refinement and anti-aliasing are discussed. Also, the way in which solid bodies are modeled is given, and bounding spheres are discussed.

The package was designed using a top-down approach. Pseudo-code showing the breakdown is given at the end of this chapter.

Optics and Ray Tracing

A reflected ray can be found by noting that the incident light angle equals the reflected angle, and that the reflection of a vector is in the plane defined by the vector and the surface normal vector (see Figure III-1).

A refracted ray can be found by the relation of Snell's law

$$n_i \sin \theta_i = n_t \sin \theta_t \qquad \text{(III-1)}$$

where θ_i is the angle that the incident ray makes with the surface normal vector, θ_t is the angle that the refracted ray makes with the surface normal vector, n_i is the index of refraction of the space the incident ray is traveling through, and n_t is the index of refraction of the space the refracted ray is traveling through. As with reflectivity, the refracted ray lies in the plane defined by the incident ray and the surface normal vector. It is possible to have n_i , n_t , and θ_i such that $\sin \theta_t$ is undefined (i.e., when $|n_i \sin \theta_i / n_t| > 1$). In this case, total internal reflection occurs, and no light is being refracted through the surface.

The intensity of light being reflected off of a surface is dependent upon the refractivity of the space on either side of the surface, as well as the angle at which the incident ray strikes the surface.

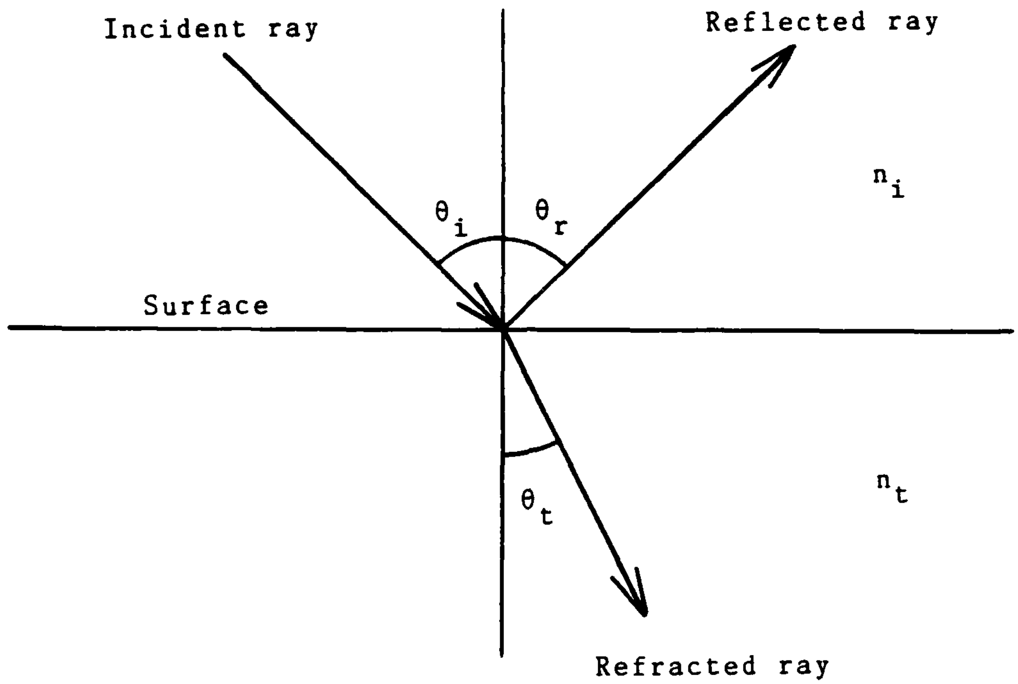


Figure III-1. Reflected and Refracted Light Rays

The reflectivity, $\bar{\mathcal{R}}$, of natural light is given by (5:45:Eq.43)

$$\bar{\mathcal{R}} = \frac{1}{2}\left(\mathcal{R}_{\parallel} + \mathcal{R}_{\perp}\right) \qquad \text{(III-2)}$$

and $\mathcal{R}_{\parallel}$ and $\mathcal{R}_{\perp}$, the reflectivity of the parallel and perpendicular component of light, are given by (5:42:Eq.33)

$$\mathcal{R}_{\parallel} = \frac{|R_{\parallel}|^2}{|A_{\parallel}|^2}$$

$$\qquad \text{(III-3)}$$

$$\mathcal{R}_{\perp} = \frac{|R_{\perp}|^2}{|A_{\perp}|^2}$$

where $A_{\parallel}$ and $A_{\perp}$ are the amplitude of the parallel and perpendicular components of the incident light and $R_{\parallel}$ and $R_{\perp}$ are the amplitudes of the parallel and perpendicular components of the reflected light.

$R_{\parallel}$ and $R_{\perp}$ are given by (5:48:Eq.56)

$$R_{\parallel} = \frac{\sin\theta_i \cos\theta_i - \sin\theta_t \cos\theta_t}{\sin\theta_i \cos\theta_i + \sin\theta_t \cos\theta_t}$$

$$R_{\perp} = -\frac{\sin\theta_i \cos\theta_t - \sin\theta_t \cos\theta_i}{\sin\theta_i \cos\theta_t + \sin\theta_t \cos\theta_i} \tag{III-4}$$

where θ_i and θ_t can be determined by Snell's law. For an incident ray normal to the surface the equation for $R_{\parallel}$ and $R_{\perp}$ can be stated as (5:41:Eq.23)

$$R_{\parallel} = \frac{n-1}{n+1} A_{\parallel}$$

$$R_{\perp} = -\frac{n-1}{n+1} A_{\perp} \tag{III-5}$$

where

$$n = \frac{n_t}{n_i} \tag{III-6}$$

The transmissivity, $\overline{\mathcal{J}}$, of natural light, which is the amount of natural light transmitted through a surface, can be found from the expression (5:45:Eq.44)

$$\bar{\mathcal{R}} + \bar{\mathcal{J}} = 1 \qquad \text{(III-7)}$$

It should be noted that, due to the symmetric nature of Snell's law with respect to the incident and refraction angles, θ_i and θ_r, can be exchanged in Eq. III-4, as well as those equations leading to Eq. III-5, causing only a sign flip in $R_{\parallel}$ and $R_{\perp}$. Because these values are squared in Eq. III-3, it does not matter as far as the calculation of $\bar{\mathcal{R}}$ and $\bar{\mathcal{J}}$ is concerned whether the ray being traced is considered to be the incident ray or the sum of reflected and refracted rays.

The sum of the contributions from reflected and refracted light is then

$$I = \bar{\mathcal{R}}\, I_r + \bar{\mathcal{J}}\, I_t \qquad \text{(III-8)}$$

where I_r is the intensity of the light before being reflected, I_t is the intensity of the light before being refracted, and I is the intensity of the resulting light, and $\bar{\mathcal{R}}$ and $\bar{\mathcal{J}}$ are as defined before.

The above equations are used only for the reflections and refractions of light not directly from light sources, but rather light reflected or refracted from other objects. The reflection of other objects off of a surface tends to affect the appearance of the surface where the incident angle equals the reflection angle. However, light from a

light source affects surfaces much more strongly. The equations for relations of light directly from light sources are as follows.

For either point or parallel light, the intensity of the light from a light source that actually reflects off of a surface (vs. that which is refracted through the surface and lost) is

$$I_l = \overline{\mathcal{R}}\, I_{ls} \tag{III-9}$$

where $\overline{\mathcal{R}}$ is calculated as before for a vector between the light source and the point on the surface, I_{ls} is the intensity of light reaching the surface, and I_l is the intensity of light actually reflected off of the surface. Ambient light is assumed, for simplicity's sake, to be in the direction of a reflected ray of whatever light ray is being evaluated, so is not treated as a light source when finding specular reflection.

For the intensities of both parallel light sources and ambient light, I_{ls} is a constant depending upon the source. For a point light source, I_{ls} can be expressed by (10)

$$I_{ls} = I_p \;/\; D^2 \tag{III-10}$$

where I_p in the intensity of light at one units distance away from the point source, and D is the distance between

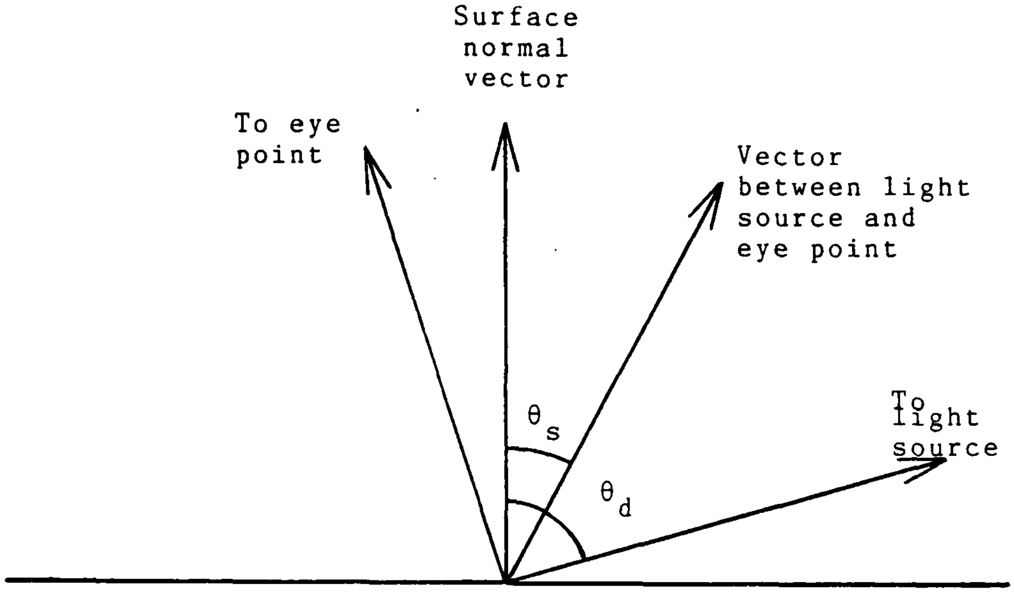


Figure III-2. Light Reflected From a Light Source to the Eye Point

the point source and the point for which I_{1s} is being calculated.

The diffuse reflection from a light source I_d can be found with the equation (10, 16, 17)

$$I_d = I_1 \, k_d \cos \theta_d \qquad \text{(III-11)}$$

where I_1 is the intensity of light from the light source, k_d is a constant, color dependent, coefficient of diffuse reflectivity of the surface, and θ_d is the angle between a vector in the direction of the light source and the surface normal vector of the surface (see Figure III-2). The amount of diffuse reflection is constant in every direction from the surface.

A simulation of the specular reflection from a light source can be found from the equation (10, 19, 25)

$$I_s = I_1 \, k_s \, (\cos \theta_s)^n \qquad \text{(III-12)}$$

where k_s is the coefficient of specular reflection, θ_s is the angle between the surface normal vector and a vector halfway between the viewer and the light source, and n is some constant describing the smoothness of the surface.

A large value of n results in less light being specularly reflected for a θ_s not equal to 0. For an n equal to 1, the equation resembles that for diffuse reflection, except that here the constant k_s does not depend upon the color of the surface. The arbitrary choice was made to define n as

$$n = \frac{1}{s} \qquad \text{(III-13)}$$

where s is a value in the range from 0 to 1 describing the smoothness of the surface. As s approaches 0, n approaches ∞, simulating a perfectly smooth surface.

The equation for light being reflected and refracted along a given ray is then

$$I = \bar{\mathcal{R}} \, I_r + \bar{\mathcal{J}} \, I_t + \sum_{i=1}^{i=ls} (I_{1_i} k_d \cos \theta_{d_i} + I_{1_i} k_s \cos \theta_{s_i})^n) \qquad \text{(III-14)}$$

where ls is the number of light sources, and everything else is as defined before. This is valid where intensities of white light and white surfaces are used. However, when colored lights or surfaces are used, the equation becomes

$$I_{red} = \bar{\mathcal{R}}\, I_{r(red)} + \bar{\mathcal{T}}\, I_{t(red)} + \sum_{i=1}^{i=ls} \left(I_{l(red)_i}\, k_{d(red)} \cos \theta_{d_i} + I_{l(red)_i}\, k_s (\cos \theta_{s_i})^n \right) \qquad \text{(III-15)}$$

where I_{red} is the intensity of red light along a given ray, $I_{r(red)}$ is the intensity of reflected red light before reflection, $I_{t(red)}$ is the intensity of refracted red light before refraction, $I_{l(red)}$ is the intensity of the red light from a light source, and $k_{d(red)}$ is given by

$$k_{d(red)} = k_{diffuse}\, c_{red} \qquad \text{(III-16)}$$

where $k_{diffuse}$ is a constant amount of didduse reflection from a surface, and c_{red} is the amount of red color in the surface. The equation for blue and green light, the other two primitives, are similar. In actuality, $\bar{\mathcal{R}}$, $\bar{\mathcal{T}}$, and k_s are color dependent also, but that color dependence will be ignored here.

The intensity and color of light traveling through a transparent or semi-transparent object is affected by the transparency and the color of the object it is traveling through. The intensity of light attenuates as it passes through an object that is not totally transparent, the loss

of intensity can be expressed as

$$I_f = I_i \, t^d \qquad \text{(III-17)}$$

where I_f is the final intensity after passing through a distance of d units of the object, and t is the transparency of the object. This affects all colors equally. The change in color due to the color of the object can be written as

$$I_{red} = I_{f_{red}} \, c_{red} \qquad \text{(III-18)}$$

where $I_{f_{red}}$ is the final intensity of red light and c_{red} is the red component of the color of the object. The equation for blue and green are similar. Notice that this does not take into consideration the change of color of light, which also depends upon the distance traveled through the object. These two equations should be used on I from Eq. III-14, with I taking the place of I_i in Eq. III-17, to find the actual intensity that reaches the end point of the ray for which the intensity and color is being found.

Additions to the UPCORE Package

Some user definable objects, attributes and parameters had to be introduced to UPCORE in order to be able to create a meaningful ray tracing package. Light sources, while not necessary for most hidden surface algorithms, were necessary

here to produce shadows and shading. Transparency and refractivity were included for completeness, since correct handling of transparent objects is one thing that ray tracing can do that most hidden surface algorithms cannot. Reflectivity and smoothness parameters were included to add needed variety to the single surface type defined by CORE.

Some other user definable parameters were needed to limit the action of the ray tracing package. Ray tree depth, color resolution, and area resolution can be used to exchange accuracy of a picture for speed of processing.

Light and distance scaling can be used to fine tune a picture, by changing the scale of the dimension or of the amount of lighting.

A description of the added features, compatible with the CORE specifications, can be found Appendix A. Appendix A also includes the description of the user functions designed in this effort, and a cross listing between the defined names and parameters and the actual Pascal names and parameters.

CORE had no representation for light sources. However, ray tracing requirees light sources to light a scene. While it would be possible to define one light source in a constant location and light amount, this was rejected as being too restrictive. Instead, user definable light sources were developed.

It was decided that different colors as well as

different amounts of light should be allowable. For parallel and point sources, which are treated as primitives, the color and amount can be set with attribute setting functions similar to those used to set primitive color in CORE. Ambient light is more of a condition than an object, so its color and intensity are set directly.

<u>Light Sources</u>. Three kinds of light sources were chosen as those being most usefull. Each has its own characteristics, and can be used for varying situations.

<u>Parallel Light Sources</u>. A parallel light source is light traveling in one direction, from an infinitely distant point. It has uniform intensity everywhere along its path.

Because of its infinite distance, any non-transparent object between the light source and a surface will block the light from that surface, so enclosed areas cannot be lit by a parallel light source.

However, a parallel light source produces a uniform light amount and direction, which is what is best for many scenes.

It was chosen to define a parallel light source by a vector defining the direction that its light traveling in.

<u>Point Light Source</u>. A point light source is an infinitely small point that is radiating light. Whereas a parallel light source is an infinite distance away from a scene, a point source is a finite distance. Because of its finite distance the intensity of light from a point light

source attenuates by an amount proportional to the square of the distance from the source. So a point farther from a point light source will receive significantly less light than a point close by.

Point light sources are difficult to work with in that their intensity is hard to adjust due to distance attenuation. However, point sources can be used to light enclosed areas, which parallel light sources cannot, and produce a more natural appearance in some settings.

It was decided to specify a point sorce by a 3-dimensional point representing its location. The amount attribute specifies the intensity at one unit distance from the source.

Ambient Light. Ambient light represents light caused by numerous reflecting off of many different surfaces. It is this light that prevents shadows from appearing black in nature. By specifying ambient light, it is possible to simplify the calculations necessary to draw a scene.

Ambient light is considered here to be of a constant amount and color everywhere. While in nature, ambient light varies in color and intensity from place to place, it was considered far too complicated to attempt anything like that here.

Ambient light is specified by color index and amount. Since ambient light is uniformly distributed, it would be redundant to have more than one ambient light source permitted.

It should be noted that the only kind of light source which will appear directly on a scene is ambient light. Due to indecision on how a light source should appear, as well as the problems apparent in the infinitely small size of a point light source and the infinite distance of a parallel light source, no reasonable representation was devised, so that only the effects of point and parallel light sources can be seen, not the sources themselves.

Point and parallel light sources are treated as primitives, so multiple examples of each are possible. Also, color and intensity are considered attributes, so by invoking the appropriate functions, it is possible to have sources of different colors and intensities.

Ambient light has no specific location or direction, and its only identifying quantities are its color and intensity, so that it is specified by specifying its color and intensity directly, instead of using the attribute values for the other light sources.

It should be noted that there are no default parallel or point light sources, and that the default ambient light amount is 0, so that by default, a scene is in total darkness. This choice was made because parallel and point sources are to be treated as primitives, which means that to specify defaults would be akin to specifying that "a green square is to be placed in the lower left-hand corner of the screen." While a user could get rid of a light source as easily as a green square, it is a nuisance to have to

remember to do so. Ambient light defaults to 0 because, in general, a scene should not be lit entirely be ambient light, and that the presence of this light source may confuse a novice user, while a black screen is a much more diagnostic symptom of having forgotten to specify a light source.

To simplify matters significantly, ray tracing does not have to terminate at a light source for a point on a surface to be lit. Instead, for each point on a surface reached through ray tracing, the contribution of light to be reflected back along the ray from each light source is calculated. The way that this is done is to trace a ray from the point on the surface in the direction of the light source. Any solids that are traveled through can reduce the amount of light reaching the surface, as well as change the color of light traveling through the solid. If a non-transparent surface is intercepted before the light source is reached, then there is no light source reaching the surface. Otherwise, the amount of light reaching the surface is calculated (see Eqs. III-10, -11, -12).

Once the amount of light reaching the surface is known, the amount of light reflected from the surface is determined by the angle at which the light hits the surface, the reflectivity of the surface, the smoothness of the surface, and the refractivity on either side of the surface (see Eqs. III-14, -15).

One more point that needs making: although the

reflection of a light source can very often be as bright as a light source itself, only actual light sources are considered when light source light is calculated. This can make a great difference when the light is to be reflected onto a diffusely reflecting surface, since no additional rays will be generated to find this light.

Additional Surface Attributes. The only surface attribute specified by CORE is the fill color, or interior color, of a polygon. The edge color, for the color of the perimeter of the polygon, is also specified, but it was decided to ignore this attribute, as the edge of a polygon is a line, and lines are being ignored in this implementation.

While default values could have been set for the following attributes, so that they were not necessary to develop a ray tracing implementation, they add a much wider variety of surface characteristics than definable with CORE.

Reflectivity. Different surfaces reflects light from them in characteristic ways. Some surfaces are "shiny" and others are not. Also, some surfaces absorb more incident light than others. The attribute that causes an object to appear "shiny," that is to reflect light at the same color and angle as was received, is called specular reflectivity. That which causes light to be reflected of the same color as the surface is called diffuse reflectivity. Specular and diffuse reflectivity are treated

here as the fraction of light that is reflected. Since, in general, surfaces do not reflect more light than is received, the specular component and diffuse component of reflectivity should not add up to more than 1. Because no surface absorbs more light than reaches it, neither component can be negative.

<u>Specular Reflectivity</u>. Specular reflectivity is what causes one's image to appear in a mirror, as well as causing high-lights from light sources on a surface. In reality, the amount of light that is specularly reflected depends upon the frequency, or color, of light. However, it simplifies the matter greatly to say that a surface specularly reflects all frequencies of light equally.

It should be noted that, with ray tracing, it is necessary to have a ray tree depth greater than 1 to have specular reflections of objects other than light sources.

<u>Diffuse Reflectivity</u>. Diffuse reflectivity is what causes a surface to appear of a particular color. A perfect mirror, not reflecting any one color more than others, can be said to be of no color. Most surfaces reflect some color more than others, and reflect that color of an amount dependent on the amount of light that reaches the surface, not on the angle at which the surface is viewed.

Such reflection, as from a wall painted with flat paint, depends both upon the color of the light and upon the

color of the surface. Only color which appears in both the surface and the illuminating light can be reflected diffusely.

While light that is reflected diffusely can come from any direction, in this implementation, only light from light sources is considered. This means that light from other objects is not considered, even when it would make a difference in the appearance of the scene. This can happen where light from a light source is reflected from a shiny surface onto the diffusely reflective surface. Another case is where a very nearby surface would cause a "blush" of its color to appear on the diffuse surface.

Because diffuse and specular reflectivity, k_d and k_s, are related in that their sum can be no greater than 1, in this implementation, they are specified together.

Smoothness. Some shiny objects reflect a sharp image while on others the image is blurred, even when the intensity of the reflected light is the same. While some models chose to think of this as a function of diffuse reflectivity, it was chosen here to think of this as separate from reflectivity, as rather the degree of small irregularities of the surface. These wrinkles affect specular reflections. The primary effect of wrinkles in this implementaion is to spread specular high-lights from light sources across a surface. The greater the degree of irregularities, the more the light is spread.

Transparency. One of the chief advantages of

ray tracing is its ability to do complicated things, like handling transparent objects, correctly. A transparent object can be thought of as one for which some light can travel through. A transparent object with a color other than white will transmit light only of that color. A transparent object with a color of black will transmit no light (see Eq. III-18).

Refractivity. Transparent objects have the additional capability of being able to refract light. This is determined by the index of refraction of the object (see Eq. III-1).

While in nature, the refractivity of an object depends on the frequency of light, allowing different colors of light to be bent by different amounts (hence, the rainbow), it is much simpler, and so was done here, to assume that all frequencies are refracted equally.

This implementation will support the refracting of rays generated in the course of ray tracing. However, because of the shortcut of finding the contribution from the light sources directly, instead of tracing back to the light sources themselves, it becomes impossible to refract light from a light source, such as is done when focusing light with a lens. It is, however, possible to change its intensity or color by having its light travel through a non-totally transparent or colored surface.

It should be noted that in addition to governing the amount that light is refracted when entering an object,

refractivity also governs the amount of light that is reflected off of a surface. A transparent object with refractivity of 1 in air or a vacuum will show no reflections off of its surface. For a higher refractivity, less light is refracted through the surface, and more is reflected off. Because of this, an object with a high refractivity will not appear totally transparent, even if it is absolutely clear.

Because the concept of refractivity can be confusing, the attributes of reflectivity and refractivity were treated so that if the transparency of an object is 0, the object is treated as though its refractivity were infinite, so that only the specular and diffuse components of reflectivity govern the amount of reflection off of the surface. For a very small value of transparency, <=1.0e-10, the surface is treated as a non-transparent one (i.e., no refracted ray is calculated) but the reflectivity is a combination of the refractivity and the diffuse and specular reflectivity attributes. With this, a non-transparent object with a known refractivity can be modeled (see Eqs. III-8, -14, -15).

The way that diffuse specular reflectivity combine with the refractivity of an object is that only that light which is reflected from the object can be specularly or diffusely reflected. Fresnel equations for unpolarized light are used to determine the amount of light reflected from an object. Fresnel equations give the amount of reflected and refracted

light as a function of angle and of refractivity (see Eq. III-4). While in actuality, unpolarized light becomes slightly polarized when reflected from a surface, this is another aspect that was ignored in the quest of simplicity.

Ray Tree Depth. Ray tracing is a very time complex algorithm. If the ray tree were generated, calculating reflected and refracted rays at each intersection with a surface, until each ray failed to intercept an object, or intercepted a light source, the algorithm would be far too complex to be practical. Instead, a limit on the depth of the ray tree is placed. The ray tree can be calculated to a depth less than the depth limit, but not to a greater depth.

The ray trace depth has a definite effect on most pictures. For example, no specular reflections, including that of ambient light, are seen off of surfaces if the depth is set to 1. For a depth of 2, reflections of surfaces will appear, but not reflections of reflections.

For this implementation, no additional rays are generated for reflections off of diffusely reflecting surfaces, so that for a scene with only diffuse surfaces, the ray tracing depth does not matter.

Light Scaling. Light scaling results in the intensity of light being divided by a scaling factor immediately before its screen color is to be determined. It is equivalent in many ways to the shutter speed of a camera. A high shutter speed is similar to a large light scaling factor in that the resulting scene appears darker. However,

for this package, it was decided that a scaling factor being too low would result in the screen color being the maximum screen brightness of the color of the light, instead of tending towards white as with photographic film. The light scale only affects light immediately before the screen color is calculated, and affects all light colors equally.

Distance Scaling. Distance scaling results in the absolute distance (i.e., that distance obtainable from the actual coordinates given by a program) being divided by a scaling factor, which is a constant value greater than 0, wherever distance is included in a calculation. Places where distance plays a part are where illumination from a point light source is calculated, and for the attenuation of light traveling through a non-totally transparent object. A distance scale applies to an entire scene, not just to some object in it. The equations where this comes into effect are III-10 and -17.

Surfaces Defined in Ray Tracing Package

Two surfaces, planar polygons and spheres, were defined in the ray tracing package. Polygons were included because that is the only surface type definable by CORE. Spheres were included to help test the package, as shall be discussed next.

Ray Tracing Package Test Bed

CORE can only define polygons. However, planar

surfaces such as polygons are not particularly good to test a ray tracing package, since planes cannot show all of the gradual shading that a curved surface can. For this reason, spherical surfaces where added to the list of surfaces that the ray tracing package can do.

However, spheres were not added to UPCORE. The main reason for not adding them was that there currently is no code in UPCORE suitable for clipping and displaying an object such as a sphere. It would not be particularly good to add a surface that could only be seen by ray tracing the scene, because one of the advantages of CORE is being able to get a quick look at a scene without polygon fill, etc. So, while spheres were added to the ray tracing package, they were not added to CORE.

In order to use the sphere to test the ray tracing package, it was then necessary to use the package without the UPCORE interface modules. Setting up spheres in the data structures without CORE is not too difficult, since spheres can be defined by their location and radius. However, defining polygons without CORE was somewhat more difficult, since much auxiliary information is generated when the polygons are received from CORE to speed up intersection calculations.

An additional advantage to working without CORE was that the size of the final compiled file was much smaller, so that less swap time was needed whenever the system swapped the job in or out of core memory.

Discussion of Picture Refinement

Ideally, at least one ray should be traced for each pixel on the screen, with additional rays being traced as needed for anti-aliasing purposes. But then, ideally, one should have a very fast computer to do calculations on.

Because of time constraints, it was not possible to ray trace individually each pixel on the screen. Instead, it was decided to start with an area somewhat larger than a pixel, and to divide that area when necessary to produce resolution down to the pixel level when it is judged necessary.

Also, ideally, the entire picture should be generated before any of it is output, so that additional anti-aliasing can be done easily. However, the core memory limits of the computer prevented this from being done.

However, care was taken to see that rays were not generated and traced more than once. This was done by storing values from preceding rows and columns, and checking to see if preceding values were in existence before generating and tracing a new ray.

Discussion of Anti-Aliasing

Whitted (25) suggests that an economic anti-aliasing scheme for ray tracing is to cause each object to be placed in a bounding sphere large enough so that if the object lies within the volume defined by the four rays defining the four

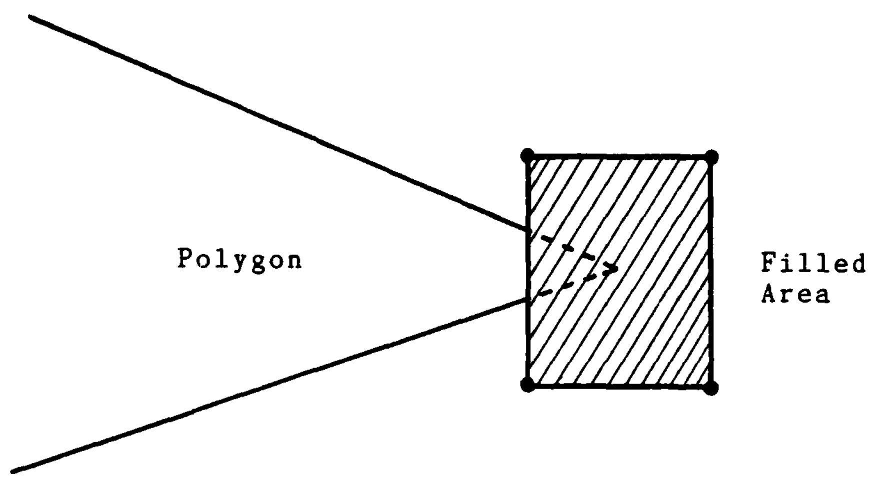


Figure III-3. Intruding Point Being Missed by the Algorithm

corners of a pixel, the bounding sphere will be intercepted by at least one of those four rays. The program will then know to subdivide the pixels around the intercepting ray until sufficient information about the fine detail is known. In addition, if the values of the color found at the four corners of the pixel are sufficiently different, again the pixel is subdivided.

While this method will work for most types of fine detail, it will not work on detail intrinsic in the shape of a surface, such as can be found with a non-convex polygon. An intruding point on a polygon can easily be missed by this algorithm, if all four rays traced for a pixel hit the polygon and the shades at each of the points of intersection is about the same (see Figure III-3).

Also, this method can become very costly for surface types that do not fill most of their bounding sphere, such

as polygons. Maximum subdivision is required everywhere within the sphere wherever the surface is not intercepted, greatly increasing the time needed to produce a picture.

A major factor in this implementation is the time constraint caused by the heavy usage of AFIT's Vax 11/780, on which this effort was being done. It was not feasible to start out with pixels equal in area to the dot size of the output device. In most cases, with a large areas of ambient light and no small detail, it is not necessary to start that small anyhow. Also, most of the surfaces being worked with do not nearly fill their bounding spheres, so to subdivide each time a bounding sphere was intersected while the object inside was missed would result in a great deal of unnecessary subdivison.

Because of this, a similar but distinct scheme is used. While it does not reduce normal aliasing well, it allows the program to start by raytracing the corners of rather large areas, knowing that in most cases the algorithm will subdivide to bring out detail within the area.

The resulting algorithm is this: 1) subdivide the area if the first object that any of the rays defining the corners of the areas strike is different from that of the others, or 2) subdivide the area if the colors at the corners are not sufficiently similar. This causes the algorithm to subdivide down to the smallest possible level for each point along the silhouette of an object and to divide to relatively small blocks when the color of the

surface is changing rapidly, but allows large areas of the same color to be filled in rapidly.

A main problem of this algorithm is that if there is some detail within the boundaries of an area that is not revealed by the corners of the area, then that detail will be lost. Common examples of this problem are where an intruding edge of a polygon or shadow is "clipped off" (see Figure III-3).

Another problem also present in Whitted's algorithm but intensified in mine is where detail of a reflection is lost. The algorithm is only forced to subdivide if the first object that the corner rays intercept is not the same. In a reflection, the reflective surface is the one which the rays strike first, and so the algorithm only subdivides the block if the surface colors due to the reflections are sufficiently different.

Solid Modeling

It should be noted that refractivity depends upon solid objects. While a surface could be modeled as an infinitely thin film, such a film will not refract light from its given path noticeably, although it could change the color of the light. Because solids are needed, and because CORE does not provide any facility for handling solids, a way of determining solids had to be decided upon.

It was decided that when a transparent surface is crossed for the first time, a solid object is being entered.

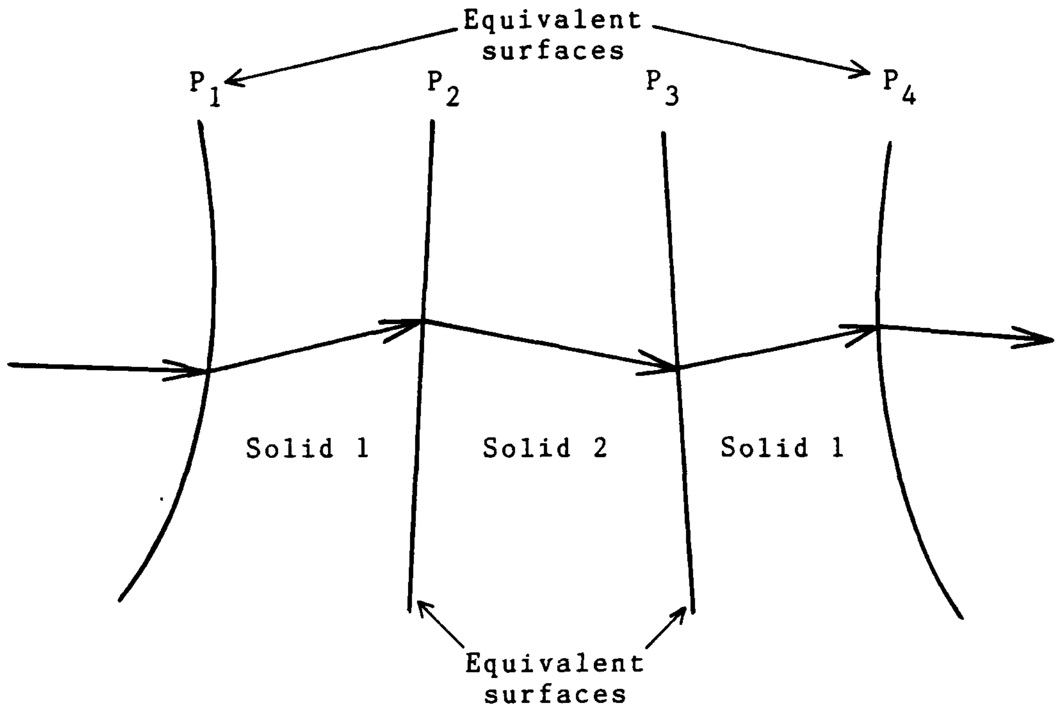


Figure III-4. Nested Solids

One remains in a solid object until a surface with same refractivity, transparency, and color is crossed, or until a surface with different characteristics is crossed. If a surface with different characteristics is crossed, it is considered to be crossing from the current solid into a nested solid, where it will remain until the appropriate matching surface is crossed to exit the nested object and enter the original one (see Figure III-4). A stack of surface characteristics is kept, allowing multiply solids to be defined, and Fresnel equations are used to calculate the amount refracted through and reflected from each surface. If in the course of defining a scene, a surface is forgotten or misplaced, incorrect pictures can be produced, as each new

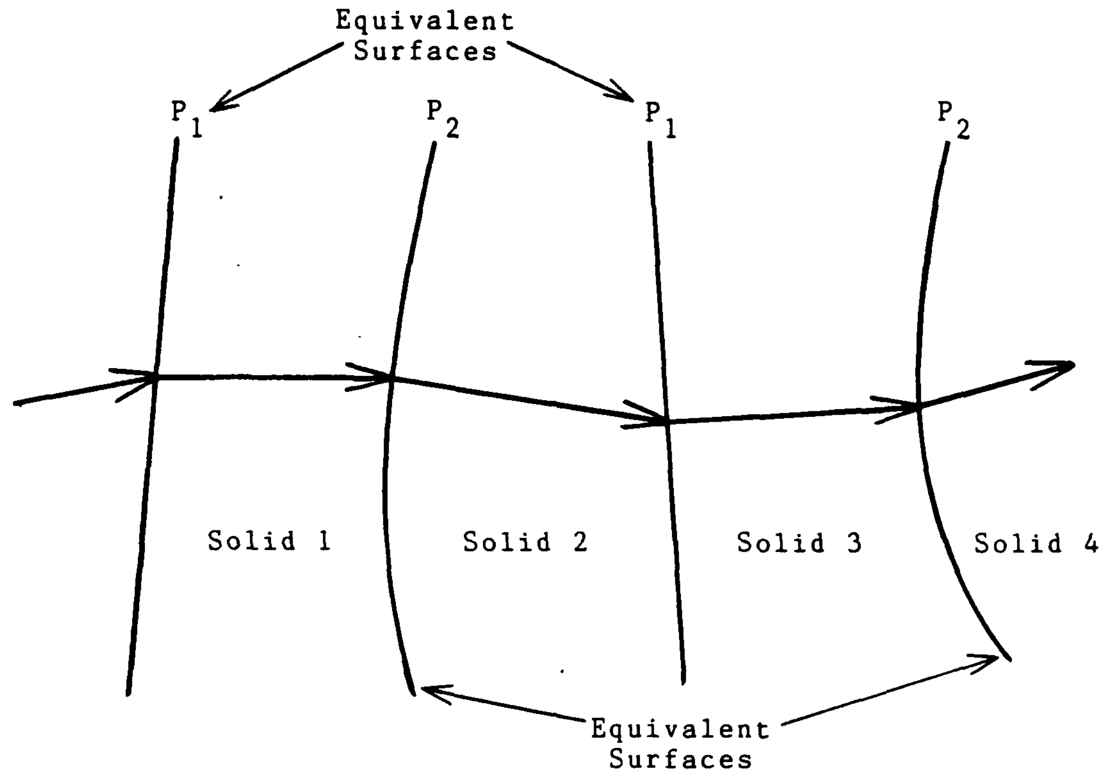


Figure III-5. Improperly Nested Solids

surface crossed adds to the confusion and depth of the solid stack (see III-5).

Also, shortcuts in defining objects, such as where intersecting polygons are used, may result in undesired hollow spaces on the interior of the solids, since the object is exited when the first appropriate surface is crossed.

Discussion of Bounding Spheres

Currently, this ray tracing package generates one bounding sphere per surface elememt. The way in which this is done for polygons is to find the bounding rectanguloid

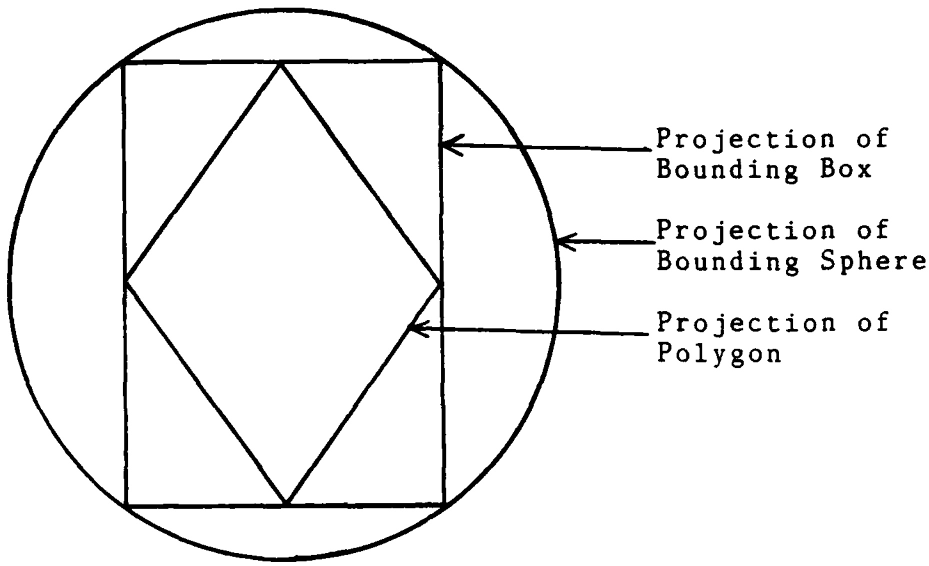


Figure III-6. Non-Minimal Bounding Sphere Around a Polygon

(i.e., the volume bounded by the six rectangles determined by the minimum and maximum values of the x-, y-, and z-coordinates of the vertices of the polygon) of the object, and then form the minimal bounding sphere around that rectanguloid. This does not, in general, provide a minimal bounding sphere. However, it is computationally fast and it works. For spherical surfaces, the bounding sphere is of course, the sphere itself.

The type of polygon for which this algorithm works least well (i.e., produces the most wasteful bounding sphere) is that in which the corners of the polygon are not near the corners of its bounding rectanguloid. An example would be a diamond shape, where its bounding sphere is a rectangle around it (see Figure III-6). Obviously, a good

deal of space is "wasted" inside its bounding sphere.

A second very wasteful type of object is an elongated polygon, where a single bounding sphere is immense compared to the true size of the polygon.

With this second sort of surface in mind, the data structures inherent in this package where designed so that multiple bounding spheres could refer to the same object. In this way, an elongated object could be encompassed by several smaller, overlapping, bounding spheres, instead of one large one. The sum of the volume encompassed by the smaller sphere would be much less than the volume of the larger sphere, reducing the number of needless attempts at intersecting the enclosed surface. Of course, this increases the number of bounding spheres to be tested for intersections with each ray, but a line-sphere intersection is computationally more efficient than intersection between a line and most other bounding surfaces which bound a volume. This package does not currently utilize the multiple bounding sphere feature of this package. However, it could be implemented easily for such surfaces as a representation of lines, where too large a bounding sphere defeats the purpose of having a bounding sphere.

An additional feature of this implementation of bounding spheres is the ability to nest a smaller bounding sphere inside a larger bounding sphere that entirely contains it. With this feature, if the larger bounding sphere is missed entirely by a ray, then the nested bounding

sphere must also be missed, so need not be checked for intersection. In some complex scenes, this feature can greatly reduce the time needed to generate the picture.

One useful feature that has not been added to this implementation of bounding spheres, but is supported by the data structures, is the concept of "empty" bounding spheres. This kind of sphere could be used to bound a small but heavily populated area of a scene from the rest of the picture. Since spheres nested within a larger sphere need only be tested for intersection if the larger sphere is intersected, this could increase the speed of the program by reducing the number of intersections to be tested for.

Introduction to Pseudo-Code

Figure III-7 is a top down pseudo-code showing how the ray tracing package was designed. Some changes were necessary upon implementation. These changes are discussed in the following section.

The actual procedure name, if any, is listed after each module. Those procedures which were absorbed in other procedures are marked by an asterisk.

Changes to Pseudo-Code Design

Ideally, the entire picture should be generated before any of it is output. However, memory constraints dictated again this. For this reasom, each area was output as it was calculated.

If the entire picture was in memory at once, it would be possible to do some retroactive corrections of such mistakes as shadow chopping and polygonal edge chopping.

```
1  Picture production (rmhidsurf)


1  Picture production (rmhidsurf)
1.1  Initialize data structures and parameters (initialize)
1.2  Find value for each pixel (raytrace)
1.3 *Output results


1.1  Initialize structures and data (initialize)
1.1.1 *Set up device related parameters
1.1.2 *Set up other parameters
1.1.3 *Develop object space (pdgetprims)


1.2  Find value for each device surface area (raytrace)
1.2.1 *Establish minimal rays to be traced
1.2.2  Trace rays (traceray)
1.2.3  Establish and trace additional rays as needed
                          (refineblock)


1.3 *Output results
1.3.1  Find color in each display surface area
                        (averagecolor)
1.3.2  Output to device through interface (pddrawrec)


1.1.1 *Set up device related parameters
1.1.1.1 *Get area resolution
1.1.1.2 *Get color resolution


1.1.2 *Set up other parameters
1.1.2.1 *Get ray tracing depth
1.1.2.2 *Get ambient light
1.1.2.3 *Get projection
1.1.2.4 *Get window
1.1.2.5 *Get view port


1.1.3  Develop object space (pdgetprims)
1.1.3.1 *Set up polygons
1.1.3.2 *Set up point light sources
```

Figure III-7. Pseudo-Code Design of Ray Tracing

1.1.3.3 *Set up parallel light sources

1.2.2 Trace rays (traceray)
1.2.2.1 Find nearest intersection for each ray (findnearint)
1.2.2.2 *Calculate reflected and refracted rays
1.2.2.3 Trace reflected ray (findrfllight)
1.2.2.4 Trace refracted ray (findrfrlight)
1.2.2.5 Find light source light (findlslight)
1.2.2.6 Sum contributions from reflected and refracted rays and light source light (addlight)

1.1.3.1 *Set up polygons
1.1.3.1.1 *Get polygons and attributes
1.1.3.1.2 Verify polygons (legitpoly)
1.1.3.1.3 *Find polygon plane
1.1.3.1.4 Find bounding sphere for polygon (detbounding)

1.1.3.3 *Set up point light source
1.1.3.3.1 *Get location
1.1.3.3.2 *Get index and amount

1.1.3.3 *Set up parallel light source Get direction
1.1.3.3.2 *Get index and amount

1.1.3.1.1 *Get polygons and attributes
1.1.3.1.1.1 *Get color
1.1.3.1.1.2 *Get reflectivity properties
1.1.3.1.1.3 *Get refractivity properties
1.1.3.1.1.4 *Get transparency properties
1.1.3.1.1.5 *Get applicable transformations

1.1.3.1.2 Verify polygons (legitpoly)
1.1.3.1.2.1 *Get polygon verticies
1.1.3.1.2.2 *Check for degeneration to line
1.1.3.1.2.3 *Find polygon plane

Figure III-7. Continued

1.1.3.1.4 Find bounding sphere for polygon (detbounding)
1.1.3.1.4.1 *Find extreme X, Y, and Z coordinate values
1.1.3.1.4.2 *Create sphere from max and min values

1.2.2.1 Find nearest intersection for each ray
(findnearint)
1.2.2.1.1 Find intersection with polygon plane (planesect)
1.2.2.1.2 See if intersection is within bounding sphere
(spheresect)
1.2.2.1.3 See if intersection is within polygon (inpoly)
1.2.2.1.4 *Choose closest intersection
1.2.2.1.5 *Of closest, choose polygon most recently created

Figure III-7. Continued

IV: Test and Evaluation

With an effort such as this, it is difficult to test the code completely. Time constraints are one reason. It takes exhaustive testing to check to see if each branch of a conditional statement has been executed, and that for every block of code, all of the extreme cases have been tried. More important, the computer time needed to run all of the extensive cases is prohibitive.

The main objective of this testing effort was to test different cases of different features of the ray tracing package, to show that the pictures produced looked as expected from optical theory. Also, some more interesting pictures were generated that show particular effects especially well.

The different cases are broken up by the effect that they are testing or illustrating.

Reflectivity

Figure IV-1 shows the relation between diffuse and specular reflectivity. In each case, the components of diffuse and specular reflectivity add up to 0.8. However, the appearance of the ball changes drastically from case to case.

In Photo IV-1(a), the diffuse component of reflectivity is 0.8 and the specular component is 0.0. In this case, the

ball has the appearance of a matte surface, with no specular reflections of the surroundings or of the light source.

In Photo IV-1(b), the diffuse component is 0.5 and the specular component is 0.3. Here, specular reflections of both the surroundings and of the light source become readily apparent. However, the red color, produced by the diffuse reflectivity of the sphere, is still strong.

Photo IV-1(c) shows weakening diffuse reflection and strengthening specular reflection as the coefficient of diffuse reflectivity is reduced to 0.3 and the specular is increased to 0.5. The red color of the sphere is much less apparent now.

In Photo IV-1(d), the red color of the sphere is no longer visible, because the coefficient of diffuse reflectivity has been reduced to 0.0. The specular reflections are very pronounced with a specular component of reflectivity of 0.8. If the specular component of reflectivity were 1.0, this sphere would be a perfect mirror.

Reflectivity and Refractivity

Figure IV-2 shows the effect of refractivity on the appearance of a totally transparent sphere in air. The depth of the ray tree is the same for all four photos.

Photo IV-2(a) shows a sphere with a refractivity of 2.4, which is equivalent to that of diamond. Note the strong specular reflection and the dimness of light

traveling through the sphere.

Photos IV-2(b) and IV-2(c) show spheres of refractivities equal to 1.5 (glass) and 1.33 (water), respectively. As the refractivity approaches that of air, the specular reflection from the light source becomes weaker and more ambient light refracts through the surface of the sphere.

Photo IV-2(d) shows a sphere with a refractivity of 1, equal to that of air. No light is reflected from its surface, and all light which strikes the surface of the sphere is refracted through. This results in an invisible sphere, although it was necessary to trace the scene to as great a depth as for the other spheres.

Figure IV-3 shows a hollow shell, totally transparent, and with a refractivity equal to that of glass. It was intended to show the effects of refractivity on a clear glass shell. Note that there are two specular reflections on this sphere. The one on the upper left-hand corner of the sphere is off of the outside of the shell, and the one in the lower right-hand corner is off of the inside of the shell. The reflection off of the inside of the shell is in a slightly incorrect position, because of the approximations used in finding light source light that is traveling through a transparent surface, which in this case is the outside of the shell.

Figure IV-4 shows the effects of a refractive sphere in a scene.

Photo IV-4(a) shows a sphere with a refractivity of 2.4, that of diamond. The refracted image of the other spheres in the background is inverted, and the images of the spheres are not distorted much. The reflections off of the sphere of the objects in the foreground are strong. The diamond sphere casts a pale shadow, but that shadow is more from the slight bluish tinge of the sphere than from its transparency.

Photo IV-4(b) shows a sphere with the same refractivity as that of glass, 1.5. Here the images of the other sphere being refracted through the glass sphere are deformed noticeably, and the reflection of the foreground scene is less distinct.

Photo IV-4(c) shows a water sphere, which has a refractivity of 1.33. Here, the image of the spheres in the background is noticeably deformed towards the edge of the sphere. Also, the amount of ambient light being refracted through the sphere is noticeably greater than that for diamond and glass.

Figure IV-4(d) shows a sphere with a refractivity of 1.005. This refractivity is close to that of air, so that this sphere has a fairly long focal length. Because of this, the images of the background spheres, being fairly close, are not inverted. They are, however, deformed slightly, especially along the perimeter of the sphere. Also, reflections off of the outside of the sphere cannot be seen.

Figure IV-5 shows the effects of refractivity on a non-transparent sphere. Here, the transparency was set to a positive value close to 0. Since the transparency was not 0, this implementation of ray tracing used the refractivity to calculate the amount of light being reflected from the surface. However, because the transparency was sufficiently close to 0, no refracted rays were generated, which sped the generation of the picture.

Photo IV-5(a) shows a sphere with a refractivity of glass (1.5). Because its refractivity is relatively low, very little light is relected diffusely or specularly, so that little of the blue coloring of the sphere shows. On this picture, the angular dependence of the Fresnel's equations can also be seen clearly. This is shown by the brightness of the specular reflection of the planar surface off of the sphere. Where the angle between the surface normal vector and the incident light is high, more light is reflected, and so the reflection appears brighter along the edges of the sphere than in the center, where the angle was lower.

In Photos IV-5(b), IV-5(c), and IV-5(d), both the brightness of the blue color of the sphere and the brightness of the specular reflections from the sphere increases as the refractivity increases. The refractivities are 3.0, 5.0, 100.0, respectively.

Figure IV-6 shows another aspect of reflectivity and refractivity.

In Photo IV-6(a), a clear diamond sphere is imbedded in a clear glass sphere. Despite the fact that both objects are totally transparent, the diamond sphere can be seen clearly inside the glass sphere, because the diamond sphere causes the light to be bent more as the light passes through it, and also reflects some light off of its surface.

Photo IV-6(b) is of the same glass sphere, but this time a magenta glass sphere has been placed inside of it. Since the refractivities of both spheres are the same, the magenta sphere would be invisible inside of the clear glass sphere if they were of the same color. As it is, light is not bent as it passes from the glass to the magenta sphere, but its color is changed as it travels through the magenta glass sphere. As an illustration of some of the inaccuracies of the approximations for light from a light source traveling through a transparent object, note that the clear glass sphere does not cast a shadow, and that the magenta sphere casts a magenta shadow. In reality, the clear glass sphere should cast a shadow, since it has a refractivity greater than 1, regardless of its transparency.

Figure IV-7 shows a scene in which a clear blue diamond cube has been placed inside of a clear blue glass cube. There is no ambient light in this scene. Instead, a white plane has been placed behind the cubes. While it is difficult to confirm the accuracy of this picture, it is rather pretty.

Smoothness

Figure IV-8 shows the effects of the smoothness attribute on an object. As is seen, the smoothness affects only the appearance of the specular reflections of light sources, not the appearance of reflections of other objects.

Photo IV-8(a) shows a sphere with the smoothness set to 0.1. The specular reflection is spread a great deal, so much that it looks as if it were in a much brighter light source. The specular reflections of the surroundings look crisp and sharp, which is one of the major flaws in this implementaion.

In Photo IV-8(b), the smoothness is set to 0.01. The specular reflection of the light source is much less, but still does not look totally accurate because of the crispness of the other reflections.

Photo IV-8(c) shows a smoothness of 0.001. Now, the specular reflection is spread over a small area only, as if the sphere were a very smooth, shiny, ball.

Photo IV-8(d) shows both an extreme case and a quirk of this implementation. Here, the smoothness is set to 0.0001. When this scene was ray traced, the specular reflection was spread over such a small area that it was missed by the algorithm. The location that it should have occupied was covered by a rectangle of the color of the surrounding area. The reflection might have been found if the area resolution had been slightly higher or lower. The color resolution probably would not have made too much difference in this

case, since it was already set to a high value.

Distance Scaling

Figure IV-9 shows the effects of distance scaling. These photos are a top view of a white plane. On the plane, twelve spheres are arranged in a circular pattern. Slightltly above the plane and off center to the left is a point light source. The light from the point light source attenuates with distance, so that the intensity of light from the source that reaches a point is related to the distance from the source.

In Photo IV-9(a), the distance scale is 7.0. With this scale value, and the original light source amount and coordinates of the scene, the spheres closer to the light source are barely lit, and those farther away are difficult to see.

In Photo IV-9(b), the distance scale is set to 9.7. Here, the lighting is a little better, with all of the spheres being visible. Notice the inner portion of the circle on the plane for where the point source light is brightest. The inner portion of the circle appears the same shade because the color corresponding to the light in that area has reached its maximum intensity, although the light itself continues to increase in intensity as it approaches the center of the circle.

In Photo IV-9(c), with a distance scale of 30.0, the circle of maximum intensity is quite large, and the

distinction between the lit and unlit halves of the sphere is sharp.

In Photo IV-9(d), the distance scale is 3000.0, and the screen is either lit to the maximum color intensity, or in total darkness from a shadow. The distinction between the lit portions of the spheres and the plane cannot be seen clearly.

The effects of distance scaling on a semi-transparent object are shown in Figure IV-10. The object in question is a sphere with the same refractivity as glass, 1.5, a transparency of 0.9, and a radius of 150 units.

Photo IV-10(a) shows the scene with a distance scaling factor of 10.0. This gives the sphere an effective radius of 15.0, far too great for light to travel through. Note, however, that the specular reflection from the light source can still be seen.

Photo IV-10(b) was produced by setting the distance scaling factor to 100.0. That causes the effective radius of the sphere to be 1.5, which is small enough for some light to travel through.

Photo IV-10(c) is the same sphere with the distance scaling factor of 1000.0, for an effective radius of 0.15. At this size, most light can travel through.

Photo IV-10(d) shows little difference from Photo IV-10(c). It was done with a distance scaling factor of 1000000.0, which reduces the effective radius of the sphere to 0.00015. At that radius and for that transparency,

almost all of the light is transmitted. The reason that the sphere appears at all is due to its refractivity. Some of the light reflects from the surface of the object, so that not all of the light enters the sphere to traverse to the other side.

<u>Ray Tree Depth</u>

Figure IV-11 shows the effects of ray tree depth on reflective surfaces.

In Photo IV-11(a), the scene is traced with a depth of one. At a depth of one, only diffuse reflections of light sources and specular reflections of point and parallel light sources are found. Not even specular reflection of ambient light is seen. Because of this, both the red and the blue ball appear fairly dark. The green ball has only diffuse reflectivity, so its appearance is unchanged by ray tree depth.

In Photo IV-11(b), the ray tree depth is increases to 2. Now specular reflection of ambient light makes both the red and the blue ball appear lighter. Reflections of the surrounding plane and of the green ball appear on both red and blue balls. Also, a reflection of the red ball appears on the blue ball, and vice versa. These reflections are dark, though, because the ray tree depth is not deep enough to find specular reflection of ambient light off of the reflections of the red and blue balls.

In Photo IV-11(c), the ray tree depth is 3. The

reflections of the red and blue balls off of each other appear brighter, and double reflections of the red ball off of the blue ball and back onto the red ball, and vice vesa, appear, though darkly.

In Photo IV-11(d), with a depth of 4, the double reflections show more color, and if the resolution were fine enough, triple reflections would begin to show. Note that throughout this series, the green ball has not changed in appearance, because it has a specular component of reflectivity equal to zero.

Color Resolution

Figure IV-12 shows the effects of color resolution on a scene containing gradual shade changes. Note that despite the color resolution, the algorithm always subdivides along the edges of the sphere. However, since the shadow is not an object in itself, the algorithm is not required to compute its boundary with a higher resolution.

Photo IV-12(a) shows a sphere done with a color resolution of 0.01. At this level, the human eye cannot discern the difference between the intensities of the adjacent pixels along the shaded area.

For Photo IV-12(b), the color resolution was set to 0.05. At this level, the difference between areas of different shade is just becoming apparent.

The color resolution of 0.15 in Photo IV-12(c) caused the areas of averaged color to become larger and more

pronounced.

Photo IV-12(d) shows the effects of a color resolution of 1.0. At this resolution, any difference in color is acceptable. Here, the shadow cast by the sphere is averaged in with the color of the plane. However, the picture is still subdivided along the perimeter of the sphere since that portion of the algorithm is not governed by the color resolution.

Area Resolution

Figure IV-13 show some of the problems and advantages of varying area resolution. This figure is of a star shaped puzzle with 48 facets, being reflected off of a highly shiney cube and off of the slightly reflective plane it has been placed on. The lines marking the bottom edges of the cube were created by leaving a gap between the bottom of the cube and the lower edges of the sides of the cube.

In Figure IV-13(a), a coarse resolution of 15 by 15 screen divisions was used. Because of this, several corners of the star puzzle are chopped off, because the algorithm did not detect that they were there. The interior of the puzzle is done correctly though. However, the interiors of the reflections of the puzzle are not necessarily calculated accurately, in addition to having the same problems as the original puzzle with points of the puzzle being missed. In particular, note the primary reflection of the puzzle off of the white plane that it is placed on. In the light pink

area of the center of the reflection, a good deal of detail was lost because the colors of the reflections of two different triangles were the same. No such problem was encountered in the original because the two triangles were defined as different objects.

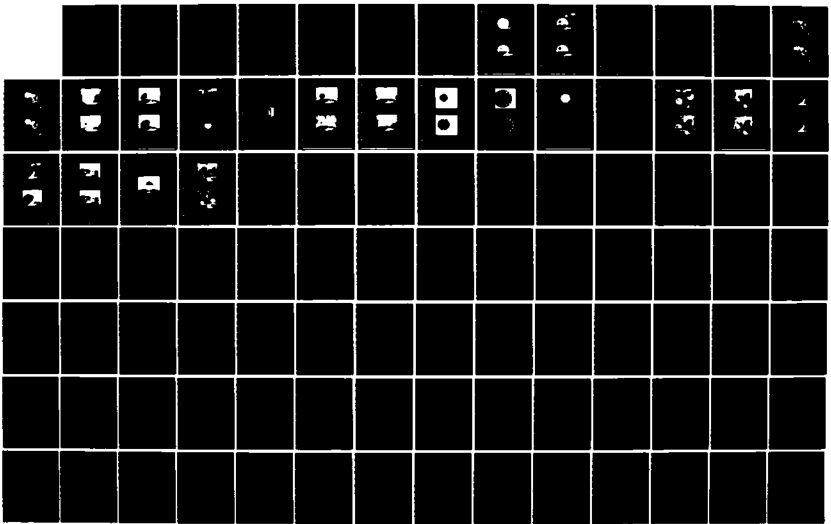
In Figure IV-13(b), a finer area resolution of 40 by 40 screen divisions was used. This eliminated most of the major trouble areas. In addition, because there were few major trouble areas that needed to be improved, and the picture contained a good deal of detail, neither the size of the output file nor the amount of time needed to create it increased much from that for the coarser resolution.

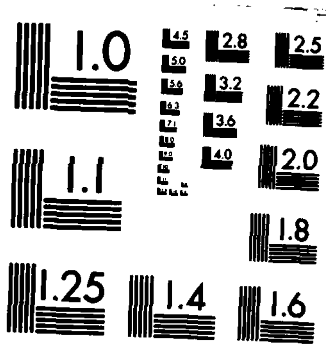
Multiple Light Sources

Figure IV-14 shows an example of multiple light sources. In this case, three light sources are shining on a white ball from three different locations. Each of the light sources is a different one of the three primary colors of light: red, green, and blue. The areas where all three lights strike evenly are white, since the three primary light colors add up to white. The area where the three shadows from the sphere overlap is a dark grey (illuminated by the ambient light) since the absence of light is black. Where two of the shadows overlap, the color of the surface is the color of the third light. Where only one shadow falls, the surface is the sum of the colors of the other two lights. Note that where the angle of the surface with

AD-A153 323 THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO A GENERAL PURPOSE..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L R SUZUKI DEC 84 AFIT/GCS/MATH/84D-6 F/G 9/2

UNCLASSIFIED NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

respect to the light source is changing gradually on the surface of the sphere, a very gradual color shift takes place.

Special Effects

In this implementation, specular reflections are of the same color as the incident light. However, many substances such as gold and copper, reflect some colors of light more than others. Figure IV-15 shows an attempt to simulate such substances.

Photo IV-15(a) shows a scene with several shiny spheres of different colors and reflectivies. Photo IV-15(b) shows the same spheres with three of the spheres now white, with their origional color being given to a thin film coating (simulated with a slightly larger, transparent, sphere) over the sphere. Now, light reflecting off of the sphere must travel through the film. This screens out some of the colors in the light. The resulting reflections off of the sphere take on the color of the film, drastically changing their appearances.

Evaluation of the Suitability of Adding an Advanced Scene Rendering Algorithm to CORE

Many changes had to be made to UPCORE in order to add the ray tracing package. Most of these changes involved adding new primitive and attribute types. Adding attributes and primitives turned out to be a simple matter, since it

was a process of making methodical additions to existing code. Major code changes were also needed to store the unclipped coordinates that the ray tracing package requires. Again, this turned out to be a matter of methodical changes.

In order for the ray tracing package to work, light sources had to be defined. These sources were added as primitives to UPCORE, rather than defining standard sources. These additional primitives caused UPCORE to depart farther from ther CORE standard, which, in a sense, is not good. Also, additional static attributes were added to polygons in UPCORE, which was another step away from the CORE standard. However, neither of these additions affected the normal operations of UPCORE. That is, UPCORE can be used in the same way it was used previously with only noticeable change being the addition of the display mode to the batching of updates. So in that sense, these additions did no real harm to UPCORE, aside from the already mentioned harm of departing more from the CORE standard. The benifits from the addition of a hidden surface algorithm to UPCORE (i.e., ray tracing) probably outweigh the harm caused to UPCORE by the addition of the algorithm.

The dynamic attributes of segments in UPCORE have not in any way been changed by the addition of light sources as primitives. For example, even though not all transformations are not applied to the unclipped coordinates of light sources correctly, the transformations are done too the ordinary UPCORE coordinates, so that if light sources

were visible to the user through ordinary (i.e., not ray tracing) UPCORE procedures, they would undergo correct transformations in the viewport. The ordinary UPCORE coordinates are also clipped correctly. Also, setting the visibility of a segment which contains a light sources to "invisible" results in that light source being "turned off" as far as the ray tracing package is concerned. This is what would be expected and desired.

Image transformations are currently not handled correctly on the unclipped coordinates that are stored for ray tracing. In fact, the only transformation that is applied to these coordinates is the user specifiable modeling transformation, aside from that of the transformation matrix borrowed from the UPCORE viewing pipeline. It is not that the other transformations are in any way difficult to use, but rather that insufficient time existed to do everything that should be done. This omission does not reflect adversely upon the suitability of interfacing a ray tracing algorithm or advanced scene rendering algorithms in general to CORE.

The only change in the procedure calls made by UPCORE was the change needed to be able to call the ray tracing package. These, changes were minor, and caused UPCORE to reflect the CORE standard more.

For these reasons, interfacing UPCORE with an advanced scene rendering technique was not too difficult. Keeping in mind that other implementations may store unclipped

parameters, and therefore be easier to interface with, it seems that interfacing this ray tracing package and a CORE graphics package, and perhaps a graphics package in general, is not difficult.

Although the interface between UPCORE and the ray tracing package was not difficult to devise, there were some problems with representing non-polygonal primitives, such as text, markers, and lines, in a way so that the ray tracing package could work with them well. The main problems were the small size and intricate shape of text characters, and the fact that lines and points are not even two-dimensional. All of these primitives could be worked with effectively if the anti-aliasing of the ray tracing were improved, and if two- or three-dimensional representations for lines and points were developed. So, the lack of ability of this ray tracing implementation to handle primitives other than polygons and spheres is the fault of this implementation of ray tracing, and not of ray tracing algorithms or of advanced scene rendering techniques in general.

Despite the difficulties encountered with lines, text, and markers, ray tracing algorithms are valid for a wide variety of surface types, such as surface patches, spheres, and others. CORE does not have these, only polygons. For a ray tracing algorithm to be of great usefulness, some of these other surface types should be present. Adding the new primitives of point light sources and parallel light sources was a simple matter, because it was decided not to display

them on the screen. Adding complicated surfaces would be more difficult, since little of the existing code would be suitable for displaying such surfaces. In this sense, it is not good to interface a ray tracing package with an implementation of CORE.

Adding the ray tracing package only added the ability to do sophisticated scenes. UPCORE still lacks the ability to produce pictures of medium quality, with hidden surface removal and perhaps simple shading. So, while adding a sophisticated scene rendering algorithm to a CORE package does produce a useful result, it will not necessarily be the best solution to the hidden surface needs of that package.

For the above reasons, it would seem that for some cases, it is useful to add advanced scene rendering capabilities to a general purpose graphics package. For UPCORE in particular, the harmful effect of moving farther away from the CORE standard was offset by the benefits from the new ray tracing ability. For specific CORE implementations in particular environments, such as where there are strict time constraints on the time needed to produce a scene, a more conventional hidden surface or shading algorithm may be in order.

As to the benefits that the ray tracing impementations gained from being interfaced with UPCORE, they are indisputable. Although CORE lacks any curved surfce types, such as spheres, making it difficult to test the ray tracing implementation, the error checking code available in a

package such as UPCORE makes it much easier to define surfaces and attributes correctly.

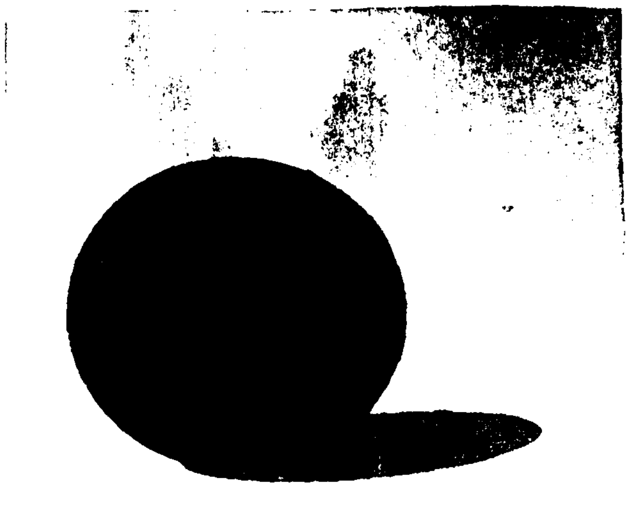


Photo (a)

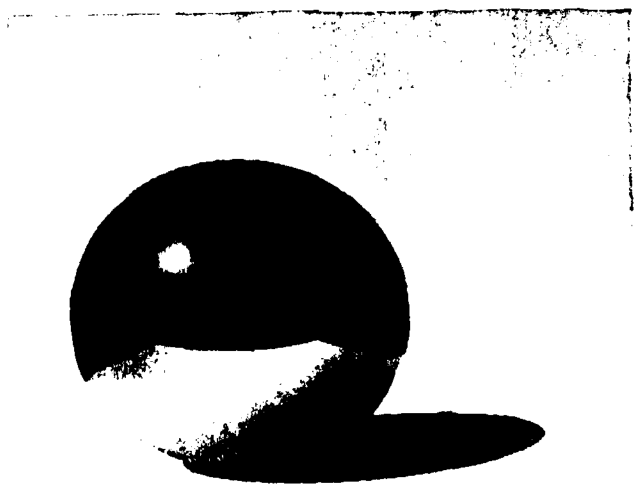


Photo (b)

Figure IV-1. Diffuse vs. Specular Reflectivity

Photo (c)

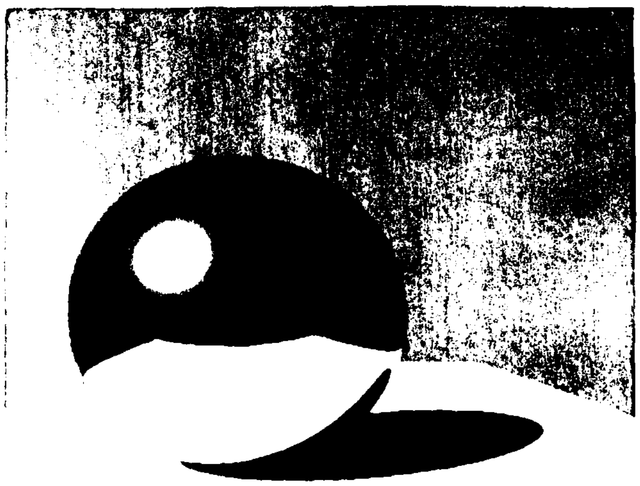


Photo (d)

Figure IV-1 continued.

Photo (a)

Photo (b)

Figure IV-2. Effects of Refractivity on the Intensity of Reflected and Refracted Light

Photo (c)

Photo (d)

Figure IV-2 continued.

Figure IV-3. Example of a Hollow, Refractive, Shell

Photo (a)

Photo (b)

Figure IV-4. Examples of a Refractive Sphere in a Scene

Photo (c)

Photo (d)

Figure IV-4 continued.

Photo (a)

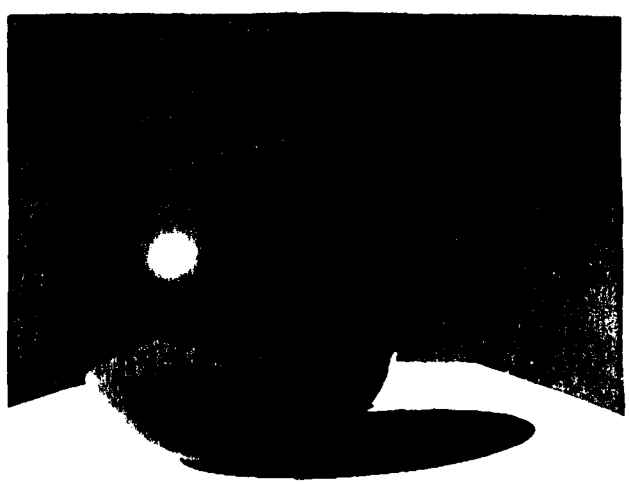


Photo (b)

Figure IV-5. Effects of Refractivity on a Non-Transparent Object

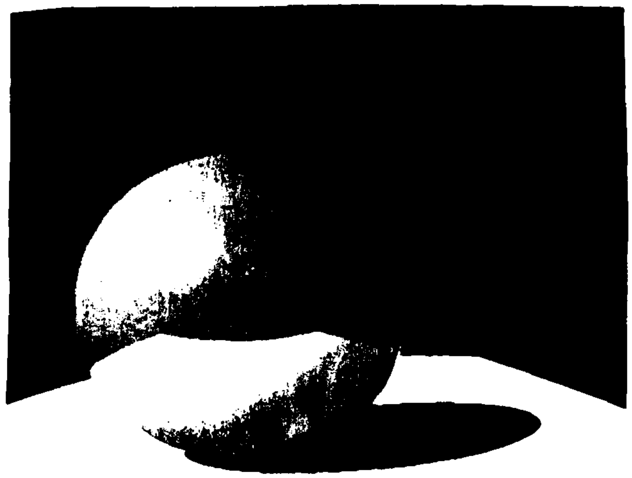


Photo (c)

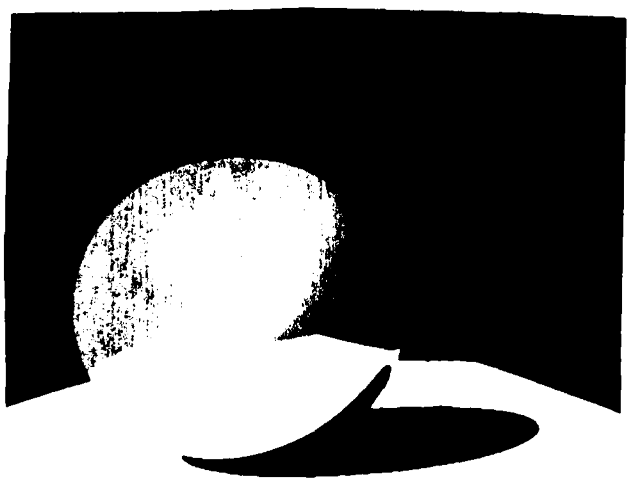


Photo (d)

Figure IV-5 continued.

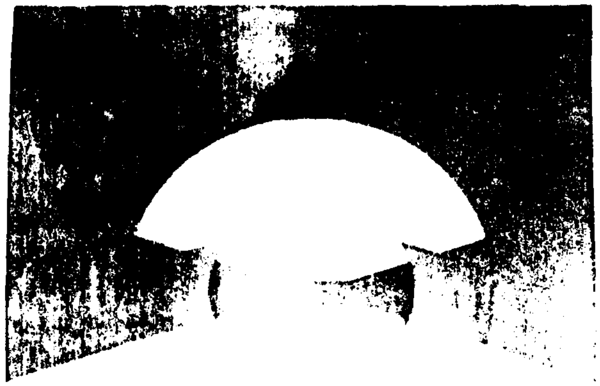


Photo (a)

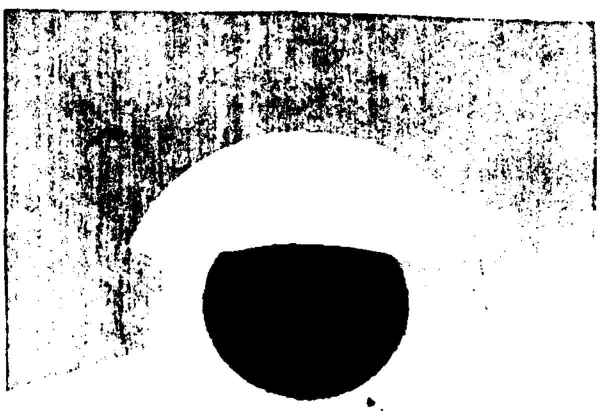


Photo (b)

Figure IV-6. Effects of Color and Refractivity in Nested Solids

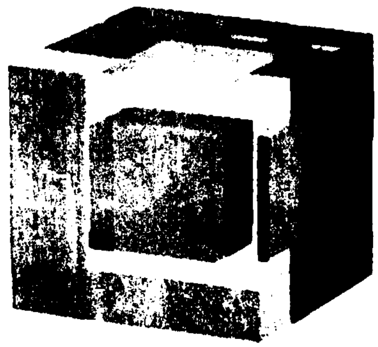


Figure IV-7. Examples of Nested Solids of the Same Color

Photo (a)

Photo (b)

Figure IV-8. Effects of Smoothness on Specular Reflection of Light Sources

Photo (c)

Photo (d)

Figure IV-8 continued.

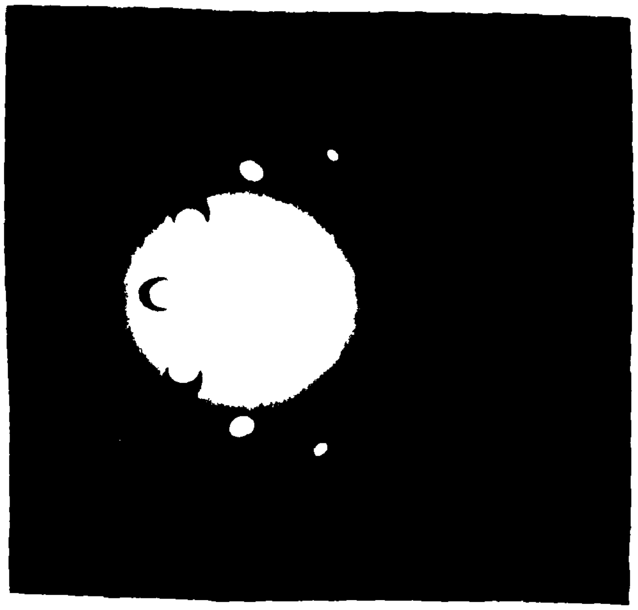


Photo (a)

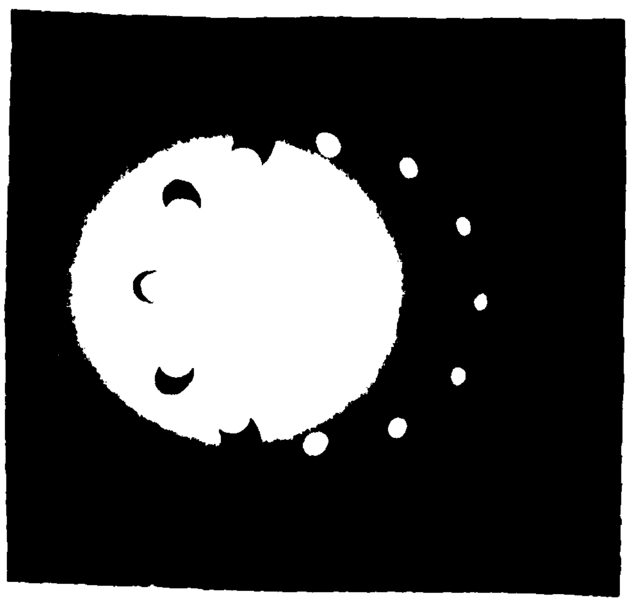


Photo (b)

Figure IV-9. Effects of Distance Scaling on Point Light Source

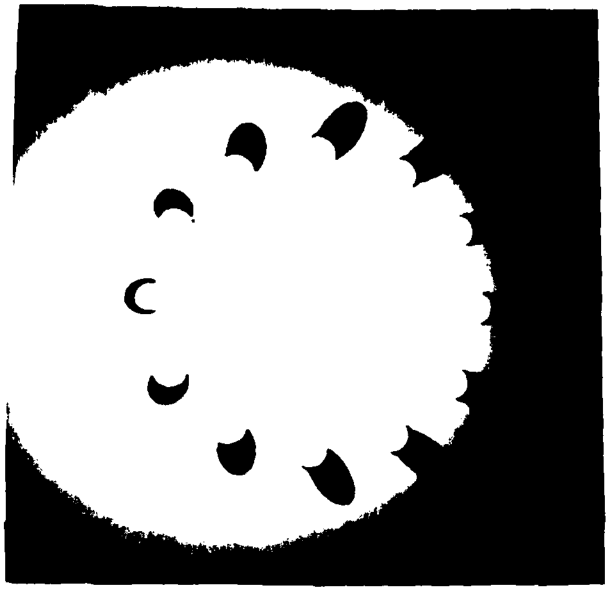


Photo (c)

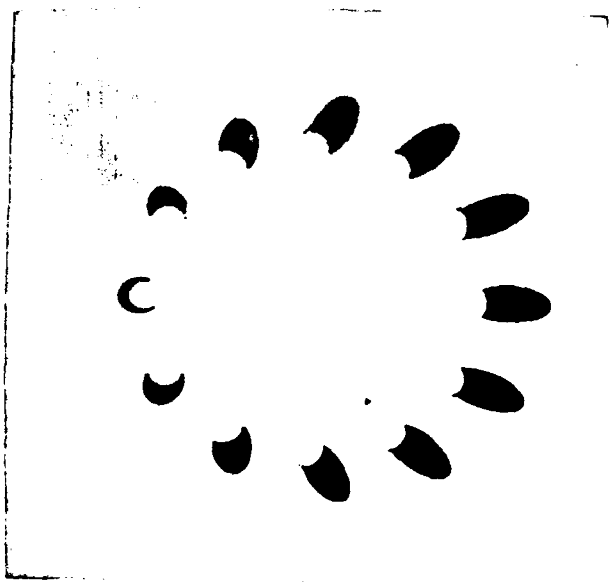


Photo (d)

Figure IV-9 continued.

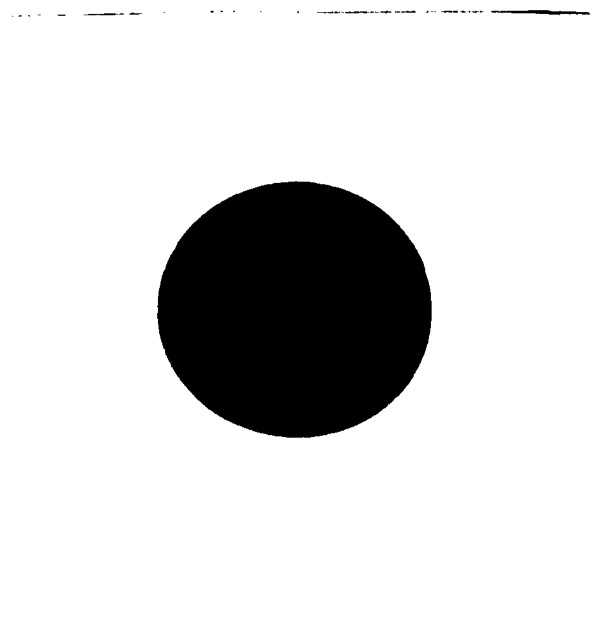


Photo (a)

Photo (b)

Figure IV-10. Effects of Distance Scaling on a Semi-Transparent Object

Photo (c)

Photo (d)

Figure IV-10 continued.

Photo (a)

Photo (b)

Figure IV-11. Effects of Ray Tree Depth on a Scene Containing Reflective Surfaces

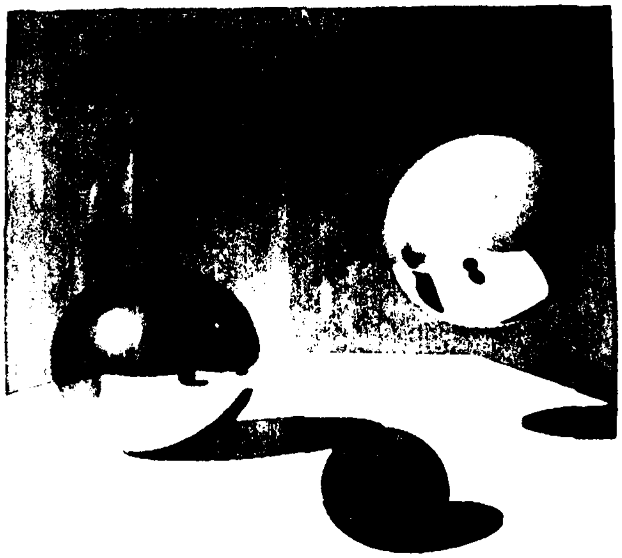


Photo (c)

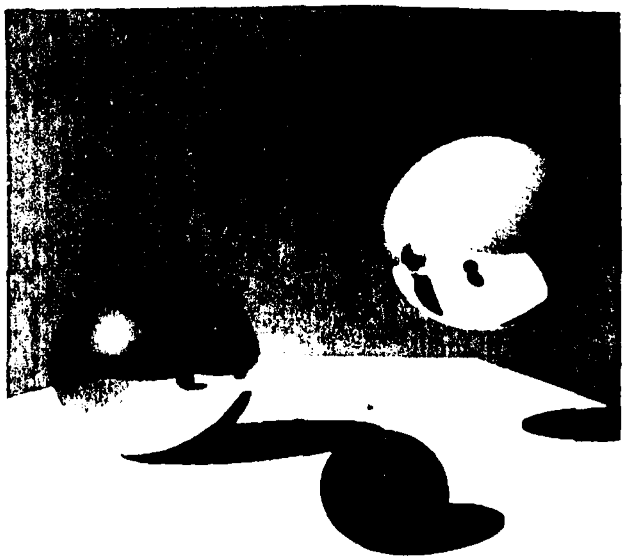


Photo (d)

Figure IV-11 continued.

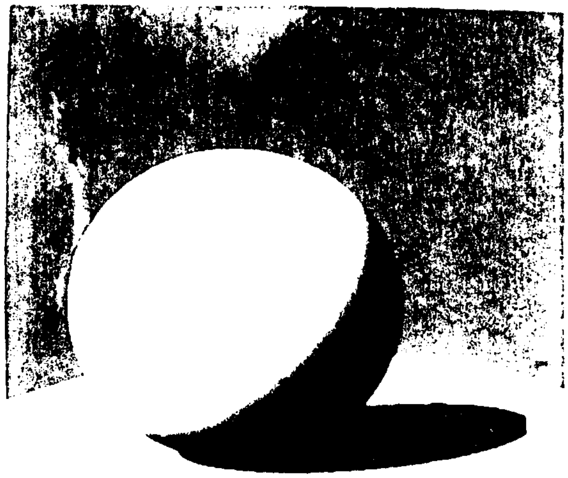


Photo (a)

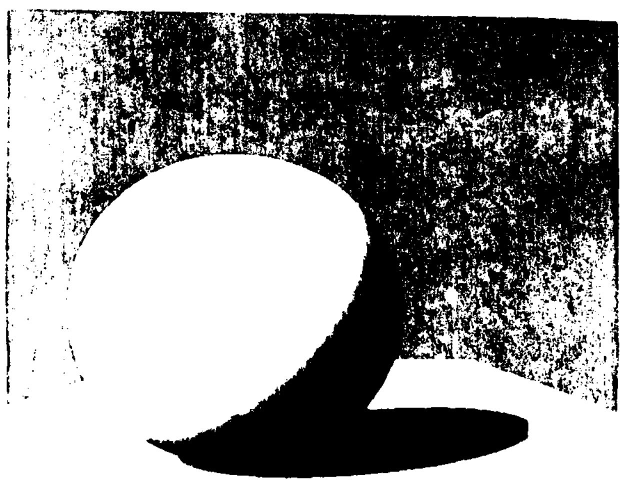


Photo (b)

Figure IV-12. Effects of Color Resolution

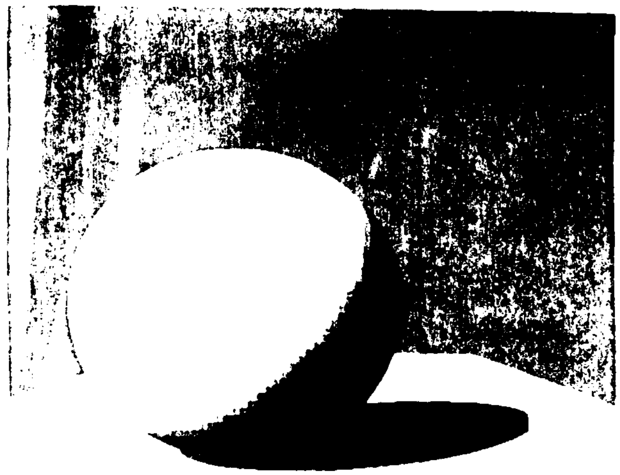


Photo (c)

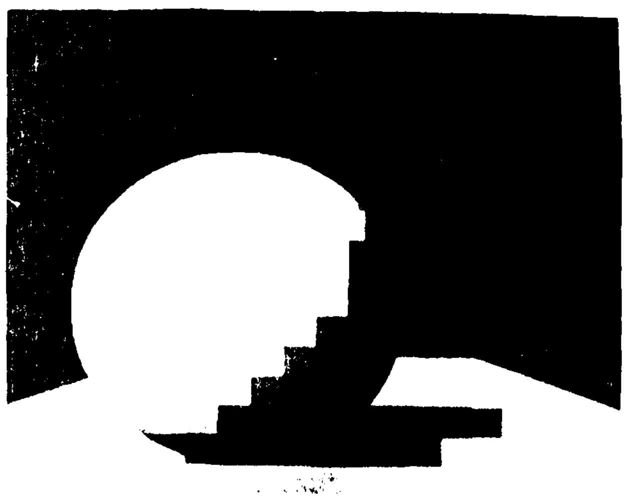


Photo (d)

Figure IV-12 continued.

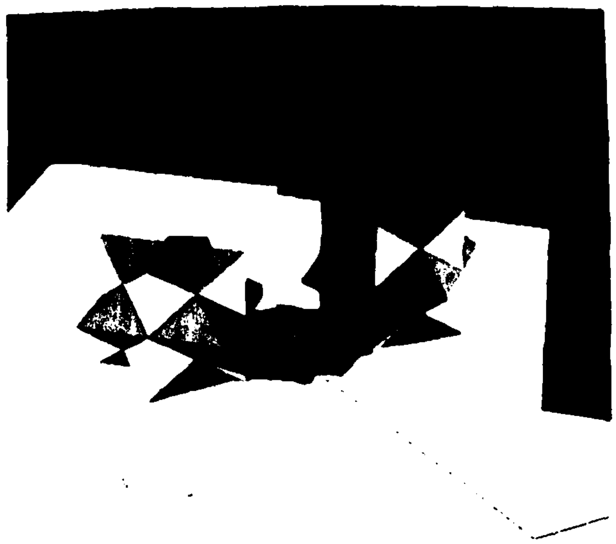


Photo (a)

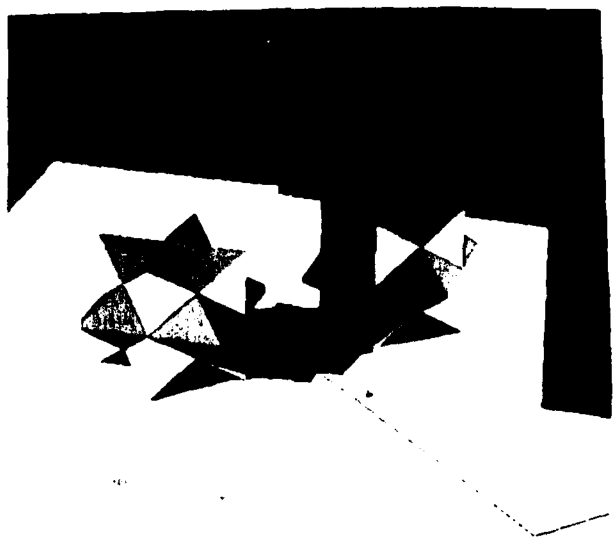


Photo (b)

Figure IV-13. Effects of Area Resolution

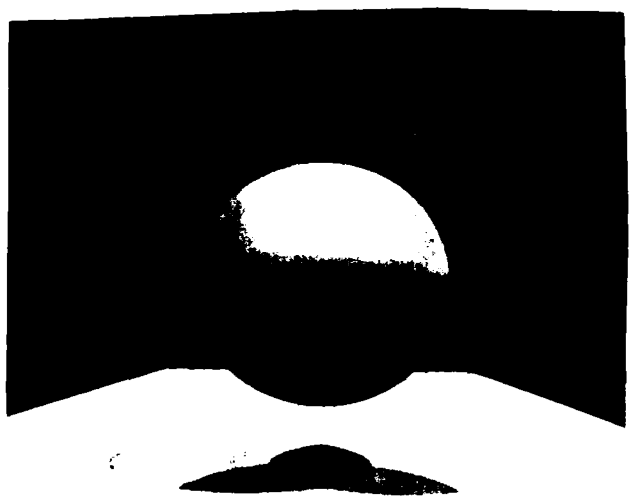


Figure IV-14. Example of Multiple Light Sources

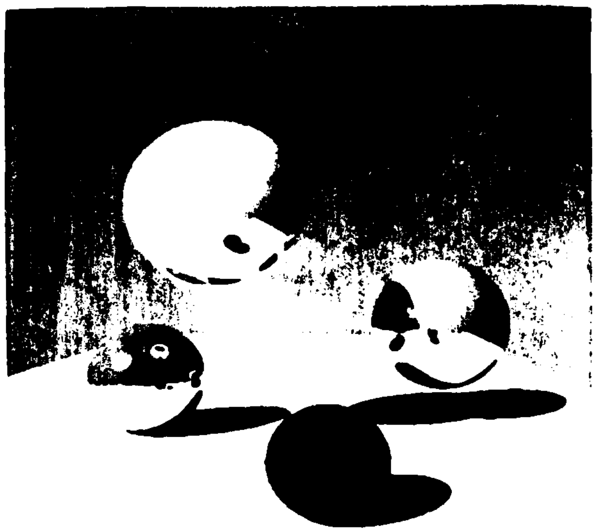


Photo (a)

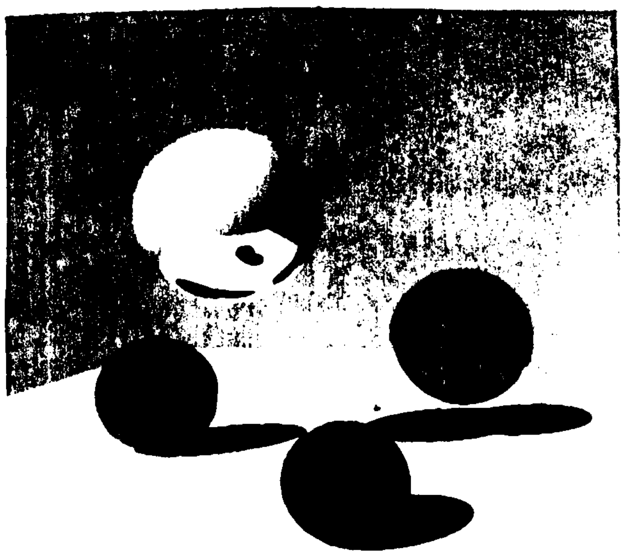


Photo (b)

Figure IV-15. Special Effects: Simulation of Color Dependence in Specular Reflection

V: Recommendations and Conclusions

Recommendations For Different Output Devices

Ray tracing can be done for any sort of output device. However, for some devices, the calculation necessary to produce the picture using ray tracing is wasted considering the quality or resolution of the output device.

It is recommended that only those devices that can output a variety of colors be used with the ray tracing package. It is also recommended that a good deal of thought be put into using this package for such devices as pen plotters, which, though capable of producing many colors, cannot produce filled areas of a given shade well. If one particularly wishes to use this package for a plotter or a similar device, it is recommended that research into ways of simulating filled areas of different shades be made.

This package could be used for suitable non-color devices by converting the final color of a given area into a shade of gray. One way to do this would be to set two of the colors to zero, and only work with intensities of the third color. However, if it is only to be used for non-color devices, it would be more efficient to modify the package to only work with intensities of white light, instead of those of red, green, and blue light.

Possible Additions to the Ray Trace Package

Additional Surface Types. The inclusion of several additional surface types would greatly enhance the ray tracing package. Particularly, with parametric patches, a wide variety of curved surfaces could be generated. Currently, curved surfaces can only be approximated with planes and spheres, and this kind of approximation is either chunky, or involves too many surfaces to be economically ray-traced. The curved surfaces definable by bi-cubic patches (10) can be used to approximate many surfaces, without the chunkiness or complexity involved with planar approximation.

Fractal surfaces can produce very natural surface appearances. Work has already been done into methods of ray tracing such surfaces, without having to generate the entire surface first (14). However, the generation and ray tracing of this sort of surface is bound to take a good deal more time than that of most other kinds of surfaces, so it is suggested that any attempt to include fractal surfaces be restricted to computer systems with the resources to handle the computation time necessary.

Also, it would be convenient to add lines, markers and text the possible surfaces, of course, modelling them as two- or three-dimensional objects with perhaps cylinders representing lines and polygons representing test and markers. But, because of aliasing problems this should not be done unless anti-aliasing is added to the package.

Surface Irregularities. A useful feature to add to

this package would be a way to map funtionally determined or predefined texture to an otherwise smooth surface. Visible deviations away from absolute smoothness occur on almost any real object, but currently, only absolutely smooth surfaces are present in the package. The simulation of microscopic irregularities is made, but this does not change the silhouette of the objects involved. The addition of texture would involve changes in procedure "detbounding," as well as procedure "intersect," to account for the altered shape of the surfaces. Considerable modification would also be reqired to the procedure "pdgetprimitives" and to the data structures, to be able to represent the texture and define the mapping of the texture to the surface to be textured.

Anti-Aliasing. This package currently possesses only the most minimal anti-aliasing features. Because of this, small detail such as corners and small objects can be missed completely, and edges of surfaces have a jagged look. However, anti-aliasing would add greatly to the amount of time needed to produce certain pictures, and so the use of it should be made optional to a user of the ray-tracing package.

Non-Uniform Color. Making color a function of position on a surface would make it possible to use fewer objects to define such objects as checker boards, where the surface could be defined by one polygon as far as the geometry is concerned, if only there were a more general way to define the color at a given point on the surface. The

CORE standard provides for only a linear interpolation of the color values at vertices. Provisions for user supplied interpolations and user supplied patterns would be useful. This would involve extensive, but methodical changes to the package. First, a type "patterntype" analogous to the type "surfacetype", giving what kind of surface pattern (solid checkerboard, polkadot, etc.) would have to be defined. Next, a record "colortype", analogous to the record "objecttype", giving the necessary fields to define each kind of pattern and its orientation on a surface, must be defined. A function must be written to return the color of an object at a point. Then, every reference to the color of an object must be replaced by a call to the approriate function Finally, default and user-specifiable orientations for the different patterns on each kind of surface must be decided on and implemented in the routine to define objects (currently procedure "pdgetprims").

Grouping. A way of specifing groupings of different surfaces in the same general area into one logical entity and placed it into one main bounding sphere could greatly reduce the time needed to raytrace scenes containing compact, but complex, objects such as the gem is a ring. As it now stands, an object such as a gem stone, with many differentfacets will have each surface in its own bounding sphere, but seldom have any of the bounding spheres nested within another sphere. Because of this, each sphere must be checked for intersections with every ray. This results in a

great increase in the amount of time needed to ray trace a scene when a complex object is introduced to the scene, regardless of whether or not the object affects anything in the scene. Currently, the package will nest only existing spheres, and will not create new spheres to bound spacially related surfaces. For best results, this feature should be user-defined. One possibility would be to enclose each segment in CORE in a bounding sphere.

Problems Encountered

Ray tracing uses far more computer resources than was realized when this effort was begun. Because of this, when running on AFIT's Vax 11/780 under UNIX, which is heavily used, it became difficult to debug and test the ray tracing implementaion. If a full understanding of the computer system limitations had been achieved before this project began, a different scene rendering algorithm would have been chosen. Figures on the computer time and elapsed time needed to produce the photographs shown is this work are in Appendix E.

Another problem encountered was that UPCORE stores clipped coordinates, which destroys the original values that the coordinates had. Since even objects not physicaly in a view port can affect the appearance of a scene (e.g., by casting shadows or being reflected off a shiny surface), it was necessary to add data structures to store the original coordinates for use in ray tracing. This increased the

memory requirements of the UPCORE package. It also detracted from the structure of the UPCORE package, and did not provide a clear interface to UPCORE.

An occasional problem occured when the logical name for a function was already used by a CORE procedure doing a similar operation on a different data structure, resulting in more artificial names being used for functions, and reducing the readability of the code. This problem could have been avoided by using a language with more information hiding, such as Ada.

Some problems were also the result of the Pascal library routines. The function 'exp(x)', which is supposed to return the value of the exponential function, would return an incorrect value when confronted with some negative arguments, but would work correctly when given the absolute value of the argument. Since the only easy way to raise a number to a power in Pascal is through the use of the "ln(x)" and "exp(x)" functions, this was an aggravating problem.

Other Recommendations

The recommendation is made that more work be done in the area of ray tracing for computer graphics. Research would be especially useful in the area of calculating the amount of light from non-point light sources. Also, specialized hardware to perform ray tracing in conjunction with a micro computer may be a fruitful path.

Another possible path would be to add an additional hidden surface algorithm to UPCORE, so that a user could decide which algorithm to choose, depending on the desired results. Another algorithm to remove hidden surfaces has already been added to UPCORE by Thomas Wailes (24), on a version of UPCORE that has been installed on another system.

A third recommendation involves the disjoint nature of ray tracing. Ray tracing is very well suited to be done with parallel processing. Because each ray is traced individually, and does not depend upon results of other rays, it is possible to trace individual rays in parallel on a multiprocessor system.

However, due to the over loaded nature of the computer resources currently available at AFIT, the recommendation is made to continue research on a less limited system.

Conclusions

Adding advanced scene rendering capabilities to a general purpose graphics package is a feasible way to improve the versatility of the actual graphics package. It provides the advantages of a well-defined, user friendly package to define surfaces and objects, and gives the sophisticated scene rendering capabilities possible with an advanced algorithm (see the section "Evaluation of the Suitability of Adding an Advanced Scene Rendering Algorithm to CORE" in Chapter IV).

However, ray tracing itself is a very time expensive

algorithm. Because of this, a package containing only ray tracing capabilities is somewhat limited in its ability to produce intermediate pictures, where the extra benefits received with ray tracing are not needed, although it may be well suited as one of several user specifiable hidden surface algorithm.

Appendix A: Discussion of User Functions Added to CORE

User Functions

The following function definitions are for user functions that have been added to the CORE package in order to be able to define desirable attributes and objects for use with the ray tracing package. They are listed and defined in a manner similar to the way CORE modules are listed and defined in the CORE specifications (21). Also, following the definitions is a cross-reference list between the defined names and the actual names and parameter types as implemented in Pascal for this effort. In addition, the cross reference list includes the page number of each defined function, since the function definitions are arranged by topic, not alphabetically.

Object Attributes. Most natural surfaces have a wide variety of characteristics that make them unique. CORE, however, defines none of them. While some characteristics, such as fuzziness, are very complicated to define and work with, others, such as transparency, can be set with one or two parameters. Of the many different characteristics of surfaces, reflectivity, refractivity, transparency, and smoothness have been chosen as most useful and easiest to handle.

Reflectivity determines the amount and color of light

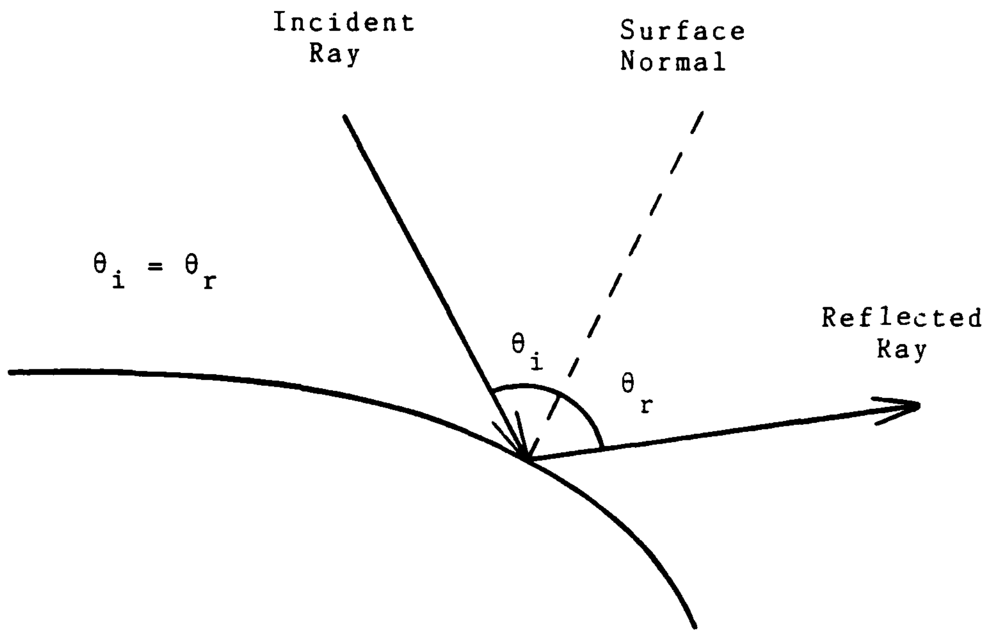


Figure A-1. Reflection of a Light Ray

that is reflected from a surface. For simplicity, reflectivity can be broken up into two components, specular reflection and diffuse reflection. Specular reflection is that in which the incident angle is equal to the reflected angle and for which the color of the light reflected is unchanged (see Figure A-1). While it is true that in reality, specularly reflected light is not always the same color as the incident light, it is an acceptable approximation for this purpose. Specular reflection is most commonly seen when looking in a mirror: one's image appears in the original colors and shape.

Diffuse reflection can be best seen when looking at a matte surface such as a smooth wall painted with a flat

paint. No image of a light source is reflected off of the wall, such as would be if it were a mirror, and the entire wall is lit by an amount that depends upon the distance from a light source. Also, the wall reflects light of a color that is both in the color of the paint and in the color of the light, so a white wall illuminated by a red light appears red instead of white. So for diffuse reflectivity, both light color and surface color are important.

It is a reasonable restraint to say that the fraction of light that is diffusely reflected and the fraction of light that is specularly reflected cannot add up to more than 1, since then in the process of reflecting, light would increase in amount, which is not typical of most real surfaces.

However, it is reasonable to say that the fraction of light specularly reflected and the fraction of light diffusely reflected can add up to a quantity less than 1, since most surfaces do not reflect all light that strikes them. A surface that does not reflect any light would appear absolutely black.

Some objects, such as glass, allow a large amount of light to travel through them. We say such objects are transparent. Obviously, not all light travels through even a fairly transparent material such as glass, or a glass cube would not cast a shadow. The fraction of light that travels entirely through one unit of a substance can be called its transparency. A transparency of 0 would result in no light

traveling through the material, while transparency of 1 would result in a substance in which all light travels through. However, even if all light travels through a unit of a substance, it can still be visible due to the fact that some of the light which strikes a surface of an object reflects away rather than refracting through, so that the full intensity does not enter the object to travel to the other side.

Transparent objects have a characteristic refractivity. The refractivity of a substance governs the angle that light is bent through when it travels from one substance to another (see Figure A-2). This bending of light is what causes a magnifying glass to be able to focus light to a single point, and is why a pencil partially in a glass of water appears to bend where it enters the water. It also affects the amount of light that is reflected from the surface of an object. The higher the refractivity of an object, the more light will be reflected from its surface (for objects in a vacuum). For this reason, a cut diamond, with a refractivity of 2.4 has more sparkle than a glass imitation, with a refractivity of 1.5. A vacuum has the lowest refractivity, a refractivity of 1. All other refractivities are greater or equal to this.

At first glance, it seems that if the refractivity governs the reflectivity of an object, then there is no need for the characteristic of reflectivity to also exist here. The reason for this is that refractivity does not dictate

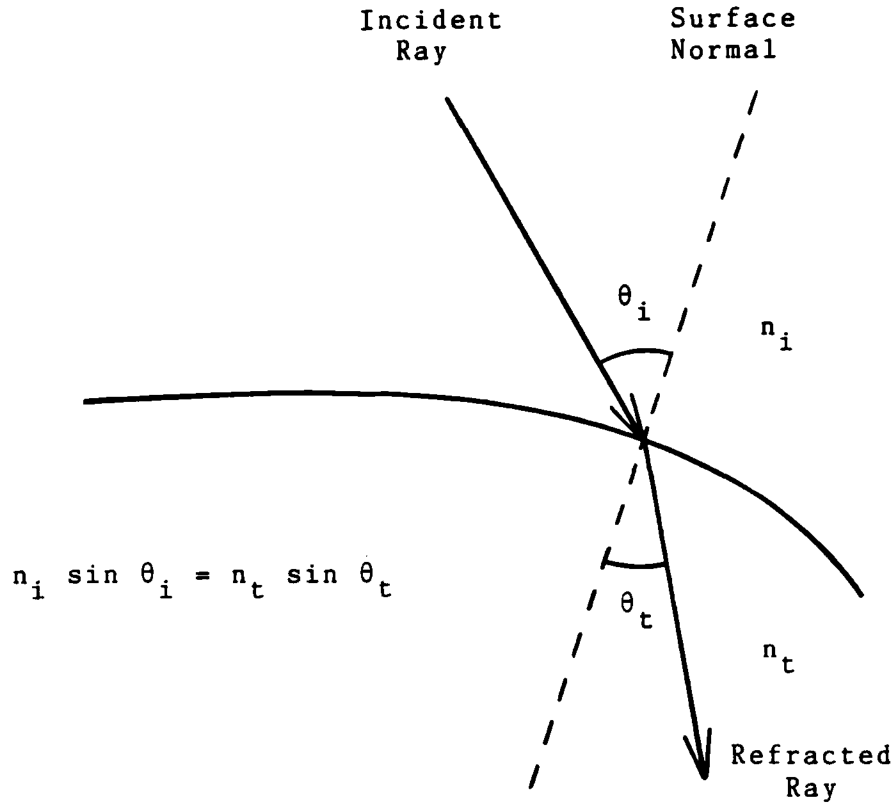


Figure A-2. Refraction of a Light Ray

the amount of light that is diffusely reflected from a surface vs. that which is reflected specularly. Objects such as frosted glass have a surface such that a good deal of light is reflected diffusely, whereas clear glass with the same refractivity reflects more light specularly. Refractivity is used with reflectivity by assuming that only light which is reflected, as determined by refractivity, can be reflected specularly or diffusely. Also, the reflectivity attribute can be used to specify the reflectivity of an object without having to work with

refractivity at all. Many people are unfamiliar with the concept of refractivity, so to force them to use it would reduce the usefulness of this package.

The way that reflectivity can be used without regard to refractivity is to set the transparency to 0 (which is the default). Then, refractivity is ignored and only reflectivity is used. A transparency close to but not equal to zero can be used to let the refractivity be used but to still simulate a non-transparent object.

Very few objects are extremely smooth, so that a reflection from an object is often blurred slightly. Usually though, the surface irregularities are too small to be felt or seen clearly, so that they do not affect the silhouette of the object they are on.

The smoothness of an object can be thought of as the degree of deviations of the surface normal from the nominal surface normal in the neighborhood of a point on the surface. A smoothness of zero would then be a surface with no deviations from the normal, i.e., one which is perfectly smooth. One with a smoothness of 1 would be one for which the surface normal varies the maximum amount possible.

The following functions are to set the new surface attributes.

SET_REFLECTIVITY (DIFFUSE, SPECULAR)

This function sets the current reflectivity attributes

to the values in DIFFUSE and SPECULAR. The parameters are real numbers between 0 and 1, inclusive, whose sum is less than or equal to 1. The default values are 1 and 0 respectively.

Errors:

1. CORE not initialized.
2. Diffuse or specular reflectivity out of range.

SET_REFRACTIVITY (REFRACTIVITY)

This function sets the current value of the refractivity attribute to the value in REFRACTIVITY. The parameter is a real number greater than or equal to 1. The default value is 1.

Errors:

1. CORE not initialized.
2. Refractivity out of range.

SET_TRANSPARENCY (TRANSPARENCY)

This function sets the current value of transparency to the value in TRANSPARENCY. The parameter is a real number between 0 and 1, inclusive. The default value is 0.

Errors:

1. CORE not initialized.
2. Transparency out of range.

SET_SMOOTHNESS (SMOOTHNESS)

This function sets the value of the smoothness

parameter to the value in SMOOTHNESS. SMOOTHNESS is a real value between 0 and 1 inclusive. The default value is 0.

Errors:

1. CORE not initialized.
2. Smoothness out of range.

The following functions are to inquire the values of the new surface attributes.

INQUIRE_REFLECTIVITY (DIFFUSE, SPECULAR)

This function copies the current reflectivity attribute values of diffuse and specular into DIFFUSE and SPECULAR.

Errors:

1. CORE not initialized.

INQUIRE_REFRACTIVITY (REFRACTIVITY)

This function copies the current value of the refractivity attribute into REFRACTIVITY.

Errors:

1. CORE not initialized.

INQUIRE_TRANSPARENCY (TRANSPARENCY)

This function copies the current value of the transparency attribute into TRANSPARENCY.

Errors:

1. CORE not initialized.

INQUIRE_SMOOTHNESS (SMOOTHNESS)

This function copies the current value of the smoothness attribute into SMOOTHNESS.

Errors:

1. CORE not initialized.

Light Sources. In nature, light is usually from radiating bodies, such as the sun or a light bulb. Such bodies are usually of finite size (i.e., not infinitely small) and radiate light in all directions from their surfaces. However, it is rather difficult to calculate illumination fron a non-point light source, such as the sun, expecially when part of it is occluded. So, it is generally easiest to work with point sources, which includes true point sources and parallel sources, rather than sources of finite size and ambient sources.

A point source can be thought of as a point in space from which light is radiating in all directions. The intensity of light is specified by the amount of light received at a distance of one unit from the source. The intensity of light from a point source decreases with the square of the distance from the source. This means that the change in illumination from a nearby source on two objects at different distances from the source can easily be seen. Also, the shadows in a scene with a point light source will always be on the side of the objects opposite of the light source, causing the shadows to radiate like spokes on a

bicycle wheel around the light source. Because this is not always the effect one wants, and because it often requires some experimentation to arrange the desired illumination level when working with point sources, it is often more convenient to work with parallel light sources. A parallel light source is equivalent to a point source located at infinity, where the amount of light received from the source is constant. Since the source is at infinity, the light rays from it are traveling in parallel, so it is easiest to think only of the direction that the light is going in, not at where it originated from. This can easily be represented by a vector in the direction of travel of the light. The amount of light from a light source is constant at every point along the path of the light.

A third sort of light is diffuse light that is not coming directly from any particular parallel or point light source, but is rather coming from all directions, from the many reflections and refractions off of surrounding objects. This is ambient light, and it is the reason why that which is in a shadow generally does not appear pitch black. In reality, ambient light in not constant everywhere, but varies a great deal from location to location. However, it is easiest to approximate it as a constant value.

Light can be thought of as having color and intensity. Natural light includes such characteristics as polarity, but such characteristics are too complicated for a simple model. Sunlight is generally white in color, but artificial light

can come in any color. Since the intensity of light from a point source decreases with the square of the distance, distance must be considered when setting up point light sources. Multiple light sources can be set up, and each one is of the amount and index that the light attributes were set at when it was
created. However, since ambient light is more of a condition than a light source, only one specification of ambient light can be made.

The following functions are to define a parallel or point light source in the current light index and amount.

POINT_LIGHT_SOURCE_ABS (X, Y, Z)

This fuction places a point light source at the location (x, y, z). CP is updated to point (x, y, z).

Errors:

1. CORE not initialized.
2. Device not initialized.
3. No open segment.

POINT_LIGHT_SOURCE_REL (DX, DY, DZ)

This function places a point light source at the point (X + DX, Y + DY, Z + DZ), where (X, Y, Z) is the CP. CP is updated to point (X + DX, Y + DY, Z + DZ).

Errors:

1. CORE not initialized.

2. Device not initialized.

3. No open segment.

PARALLEL_LIGHT_SOURCE_ABS (X, Y, Z)

This function establishes a parallel light source located at infinity whose light is traveling in the direction of the vector (X, Y, Z). Current point (CP) updated to point (X, Y, Z).

Errors:

1. CORE not initialized.

2. Device not initialized.

3. No open segment.

4. Vector is of length 0.

PARALLEL_LIGHT_SOURCE_REL (DX, DY, DZ)

This function establishes a parallel light source located at the direction of the vector (X + DX, Y + DY, Z + DZ), where (X, Y, Z) is the CP. CP is updated to point (X + DX, Y + DY, Z + DZ).

Errors:

1. CORE not initialized.

2. Device not initialized.

3. No open segment.

4. Vector is of length 0.

The following function is to set the ambient light amount and index.

SET_AMBIENT_LIGHT (LIGHT_INDEX, LIGHT_AMOUNT)

This functon sets the amount of the ambient light present. LIGHT_INDEX is a non-negative integer, LIGHT_AMOUNT is a non-negative real value. The default values are 1 and 0, respectively.

Errors:

1. CORE not initialized.
2. INDEX out of range.
3. AMOUNT out of range.

The following functions are to inquire the amount and index of the ambient light.

INQUIRE_AMBIENT_LIGHT (INDEX, AMOUNT)

This function copies the current ambient light amount and index into AMOUNT and INDEX, respectively.

Errors:

1. CORE not initialized.

The following functions are to set the parallel and point light source attributes.

SET_LIGHT_AMOUNT (LIGHT_AMOUNT)

This procedure sets the amount of light for any parallel or point light sources to be defined to the value in LIGHT_AMOUNT. LIGHT_AMOUNT is a real value between 0

and 1 inclusive. The default is 0.

Errors:

1. CORE not initialized.

2. AMOUNT out of range.

SET_LIGHT_INDEX (LIGHT_INDEX)

This procedure sets the index of light color or intensity for any parallel or point light sources to be defined to the value in LIGHT_INDEX. LIGHT_INDEX is a non-negative integer. The default is 0.

Errors:

1. CORE not initialized.

2. INDEX out of range.

The following functions are to inquire the parallel and point source attributes.

INQUIRE_LIGHT_AMOUNT (LIGHT_AMOUNT)

This function copies the current light amount attribute into LIGHT_AMOUNT.

Errors:

1. CORE not initialized.

INQUIRE_LIGHT_INDEX (LIGHT_INDEX)

This function copies the current light index attribute into LIGHT_INDEX.

Errors:

1. CORE not initialized.

Scaling Factors. It is sometimes desirable to be able to change the scale of a scene without having to change the value of every coordinate in the scene. It can also be useful to change the amount of light in a scene without having to change the values of all the light sources. For these reasons, scaling factors for light and for distance are introduced.

Distance scaling can be very useful for two reasons. The first reason is that the amount of light that travels through an object depends on the thickness of an object, as well as its transparency. By dividing the overall dimensions of a scene by 10, including those of the window, it is possible to reduce the effective thickness of an object by a factor of 10, which can greatly change the appearance of a scene, even though all of the objects appear the same size on the screen. The second reason is that light from point sources decreases with distance, so by scaling the distance, the light source can be made to appear brighter or dimmer.

Light scaling can be used for the same reasons that shutter speed is adjusted on cameras; to avoid over or under exposure. Scaling light before the final color is determined can make a scene brighter or darker, without having to change light source amounts.

In either case, the scaling factor is the value that

the light or distance in divided by to produce the desired effect (i.e., a distance scale of 10 would divide the dimensions of the scene by 10). For the distance scaling factor, wherever distance is involved in a formula used in ray tracing, that distance is divided by the distance scaling factor. For light, the division by the light scaling factor is only done before the final color is output.

The following functions are to set the scaling factors.

SET_LIGHT_SCALING_FACTER (LIGHT_SCALE)

This function sets the amount that all light is scaled by to the value in LIGHT_SCALE. LIGHT_SCALE is a positive real value. The default is 1.

Errors:

1. CORE not initialized.
2. LIGHT_SCALE not positive.

SET_DISTANCE_SCALING_FACTOR (DISTANCE_SCALE)

This function sets the amount that all distances are scaled by to the value in DISTANCE_SCALE. DISTANCE_SCALE is a positive real value. The default is 1.

Errors:

1. CORE not initiaized.
2. Scaling factor not positive.

The following functions are to unquire the scaling factors.

INQUIRE_LIGHT_SCALING_FACTOR (LIGHT_SCALE)

This function copies the current light scaling factor into LIGHT_SCALE.

Errors:

1. CORE not initiaized.

INQUIRE_DISTANCE_SCALING_FACTOR (DISTANCE_SCALE)

This function copies the current distance scaling factor into DISTANCE_SCALE.

Errors:

1. CORE not initialized.

Resolution. Often, an output device will have a limited color capability, or a picture is mostly one or two solid colors that one would wish to be filled in quickly. If an output device has a limited number of colors, it is needless to try to refine the color of an area or pixel any more than to the limits of the device. In other words, a low color resolution is desired.

The resolution of the color of an area can be defined as the largest difference that is acceptable between the colors at the corners of the area, if the area is to be all one color. Since the components of color range between 0 and 1 in the RGB color model, the way that color resolution

is defined here is the maximum acceptable difference between the corresponding components of two colors to be considered the same color, and is of course a value between 0 and 1.

Since ray tracing is so time consuming, it is often desirable to specify a fairly large area as the largest area that can be filled in at once. This allows larger areas of the picture to be filled in quickly, with the hopes that the ray tracing algorithm will do the correct subdivisions of area where smaller areas are appropriate. In other words, trade quality for time. The way that this area is determined here is to specify the number of divisions of the viewport in the x- and y-directions. So, a resolution of 10 by 20 would divide the width w of the viewport by 10 and the height h of the viewport by 20, within the limits of the output device. The largest possible area that could be filled at once would be a rectangle w/10 by h/20. When mapped to the device screen, these values are truncated to the nearest integer, with the pixel size of the screen being the limiting factor on the smallest size.

The following functions are to set the area and color resolution.

SET_RESOLUTION (X_RESOLUTION, Y_RESOLUTION)

This function sets the minimum resolution of the ray tracing to X_RESOUTION divisions by Y_RESOUTION divisions. X_RESOLUTION and Y_RESOLUTION are

positive integers. The defaults depend on the output device used.

Errors:

1. CORE not initiaized.
2. Resolution not positive.
3. Resolution greater than screen capabilities.

SET_COLOR_RESOLUTION (COLOR_RESOLUTION)

This function sets the minimum color resolution to the value in COLOR_RESOLUTION. The default for resolution is device dependent. COLOR_RESOLUTION is a real number between 0 and 1, inclusive.

Errors:

1. CORE not initialized.
2. Color resolution out of range.

The following functions are to inquire the area and color resolution.

INQUIRE_RESOLUTION (X_RESOLUTION, Y_RESOLUTION)

This function copies the current minimum resolution into X_RESOLUTION and Y_RESOLUTION .

Errors:

1. CORE not initiazed.

INQUIRE_COLOR_RESOLUTION (COLOR_RESOLUTION)

This function copies the current value of the color

resolution attribute into COLOR_RESOLUTION.

Error:

1. CORE not initialized.

Ray Tracing Depth. The nature of ray tracing is such that simulated light rays are traced backwards through several reflections or refractions until no more objects are encountered or a limit on the depth of the ray tree is reached. As the light ray is reflected from a transparent object, a new ray is generated as the refracted portion of the ray, so the number of rays being traced can increase with the depth of the tree. Obviously, these reflections and refractions take time. Because of this it is appropriate to be able to set a limit on the number of reflections or refractions a ray can go through. This limit is the maximum depth of the ray tree. It can be any positive integer within reason, i.e., the ray tracing algorithm is recursive, and a limit too high can result in a program using its entire allotment of memory. However, in order to get reflections off of a shiny surface or refractions through a transparent surface, a depth greater than one is required. If there are no shiny surfaces or transparent objects, though, a depth of 1 is sufficient. For most scenes, a limit of 5 should be more than enough.

The following function is to set the ray tree depth.

SET_RAY_TRACE_DEPTH (DEPTH)

This function sets the current value of the ray-tree depth to the value in DEPTH. DEPTH is a positive integer. The default is 1.

Errors:

1. CORE not initialized.
2. Depth out of range.

The following function is to inquire the ray tree depth.

INQUIRE_RAY_TRACE_DEPTH (DEPTH)

This function copies the current depth of ray tracing into DEPTH.

Errors:

1. CORE not initialized.

<u>Cross Reference Index</u>

The following is a cross reference list between the defined names of the functions and the actual procedure names and parameter types used when implemented. Because the function definitions are arranged by subject instead of alphabetically, the page number that each function is defined on is listed also.

Two functions, INQUIRE_DISPLAY_MODE and SET_DISPLAY_MODE were defined in the raster extentions to the CORE standard (21). They are necessary to be able to invoke

the hidden surface removal. Since UPCORE had no hidden surface removal, these functions were not implemented at that time. The two functions were implemented in this effort, however, so the names and parameter types assigned during implementation are given here, although the functions are defined only in the CORE standard. The two procedures are indicated by asterisks.

INQUIRE_AMBIENT_LIGHT (LIGHT_INDEX, LIGHT_AMOUNT)

inqaml (var index : integer; var amount : real)

Page A - 13

INQUIRE_COLOR_RESOLUTION (COLOR_RESOLUTION)

inqcres (var colorres : real)

Page A - 19

*INQUIRE_DISPLAY_MODE (MODE)

inqdmode (var mode : integer)

INQUIRE_DISTANCE_SCALING_FACTOR (DISTANCE_SCALE)

inqdscale (var dscale : real)

Page A - 17

INQUIRE_LIGHT_AMOUNT (LIGHT_AMOUNT)

inqltamt (var amount : real)

Page A - 14

INQUIRE_LIGHT_INDEX (LIGHT_INDEX)

inqltndx (var index : integer)

Page A - 14

INQUIRE_LIGHT_SCALING_FACTOR (LIGHT_SCALE)

inqlscale (var lscale : real)

Page A - 17

INQUIRE_RAY_TRACE_DEPTH (DEPTH)

inqdepth (var depth : integer)

Page A - 21

INQUIRE_REFLECTIVITY (DIFFUSE, SPECULAR)

inqrfl (var diffuse, specular : real)

Page A - 8

INQUIRE_REFRACTIVITY (REFRACTIVITY)

inqrfr (var refractivity : real)

Page A - 8

INQUIRE_RESOLUTION (X_RESOLUTION, Y_RESOLUTION)

inqres (var resx, resy : integer)

Page A - 19

INQUIRE_SMOOTHNESS (SMOOTHNESS)

inqsmooth (var smoothness : real)

Page A - 9

INQUIRE_TRANSPARENCY (TRANSPARENCY)

inqtrans (var transparency : real)

Page A - 8

PARALLEL_LIGHT_SOURCE_ABS (X, Y, Z)

palsabs (x, y, z : real)

Page A - 12

PARALLEL_LIGHT_SOURCE_REL (DX, DY, DZ)

palsrel (dx, dy, dz : real)

Page A - 12

POINT_LIGHT_SOURCE_ABS (X, Y, Z)

polsabs (x, y, z : real)

Page A - 11

POINT_LIGHT_SOURCE_REL (DX, DY, DZ)

polsrel (dx, dy, dz : real)

Page A - 11

SET_AMBIENT_LIGHT (LIGHT_INDEX, LIGHT_AMOUNT)

setaml (index : integer; amount : real)

Page A - 13

SET_COLOR_RESOLUTION (COLOR_RESOLUTION)

setcres (colorres : real)

Page A - 19

*SET_DISPLAY_MODE (MODE)

setdmode (mode : integer)

SET_DISTANCE_SCALING_FACTOR (DISTANCE_SCALE)

setdscale (dscale : real)

Page A - 16

SET_LIGHT_AMOUNT (LIGHT_AMOUNT)

setltamt (amount : real)

Page A - 13

SET_LIGHT_INDEX (LIGHT_INDEX)

setltndx (index : integer)

Page A - 14

SET_LIGHT_SCALING_FACTOR (LIGHT_SCALE)

setlscale (lscale : real)

Page A - 16

SET_RAY_TRACE_DEPTH (DEPTH)

setdepth (depth : integer)

Page A - 21

SET_RESOLUTION (X_RESOLUTION, Y_RESOLUTION)

setres (resx, resy : integer)

Page A - 18

SET_REFLECTIVITY (DIFFUSE, SPECULAR)

setrfl (diffuse, specular : real)

Page A - 6

SET_REFRACTIVITY (REFRACTIVITY)

setrfr (refractivity : real)

Page A - 7

SET_SMOOTHNESS (SMOOTHNESS)

setsmooth (smoothness : real)

Page A - 7

SET_TRANSPARENCY (TRANSPARENCY)

settrans (transparency : real)

Page A - 7

Appendix B: Guide to Common Modifications of the Ray Tracing Package

Two changes are likely to be necessary on a routine basis to enhance the usefulness of the ray tracing package. The first change is the addition of new surface types to the package. Since the package currently can only work with planar polygons and spheres, the ability to add new surface types is very desirable. The second likely change is to interface the ray tracing package with a graphics package other than UPCORE. A more complete implementation of CORE or a more sophisticated graphics package could profit from the addition of ray tracing capabilities to a greater extent than UPCORE has.

This discussion assumes a knowledge of Pascal, and it is recommended that these modifications only be attempted by someone familiar with Pascal.

Modifications Needed to Add New Surface Types to the Ray Trace Package

Currently, the ray tracing package can only work with two surface types: planar polygons and spherical. This limits the practicality of the package, since most everyday objects consist of a wider variety of curves and textures. With this in mind, the attempt has been made to make the addition of new surface types to the ray tracing package a

straight forward process. The following is a guide to the addition of new surface types.

The new surface type must be given a unique name. Because of the limits of some versions of Pascal, the first 6 characters of the name should be different from those of any other surface type name. The name also must not include any of the special symbols of Pascal. The name must then be included in the definition of the type "surfacetype."

The information necessary to define a surface of the new surface type must be determined, and the fields needed to hold this information must be added to the definition of "objecttype."

An algorithm to define a bounding sphere or spheres for any surface of the new surface type must be developed, and implemented in the procedure "determineboundingspheres." While it is not necessary to find the minimal bounding sphere for each example of the new surface, a minimal bounding sphere will, in general, speed the processing time of a scene containing such a surface.

An algorithm for intersecting a line with any surface of the new type must be developed and implemented in the procedure "intersect." Caution should be exercised to ensure that cases in which the line intersects the bounding sphere but not the object contained within are handled correctly, as well as those where the line intersects the surface, but in a negative direction from the origin of the light ray. The algorithm must also be able to find the

closest point of intersection from the origin of the ray that the line represents, for cases in which it is possible to have more than one intersection between a ray and a surface. Also, any information which, while not necessary to the definition of the surface, but helpful to the computational speed of an implementation of the algorithm, should be included in the definition of "objecttype."

An algorithm for determining the surface normal vector at a given point on a surface must be developed, and implemented in the procedure "findsurfacenorm." The surface normal vector must be of length 1 and in such a direction that the dot product between the incident ray and the vector is negative.

Procedure "terminate" must be modified so that it can dispose of the dynamic memory allocated to surfaces of the new type. If the new surface is implemented with multiple bounding sphere capability, then a method must be found into how to only dispose of the allocated surface memory once, instead of once for each bounding sphere.

Changes Needed to Add the Ray Trace Package to a New Graphics Package

In the hope that someone will someday want to implement this package with a package other than CORE, the attempt has been made to minimize the dependence of the ray tracing package upon its attending graphics package. In fact, a daring programmer can use this package without any attending

graphics package at all. However, without the error checking portions of a graphics package, the slightest mistake can lead to rather unusual results. The recommendation is therefore made to use the package with some form of object defining package, even if it is only a specialized set of procedures for one application only.

The following instructions should help with the installation of this package with another graphics package.

Any new surface types must be added to the ray tracing package, as discussed previously. The current limitation of the ray tracing package in regard to anti-aliasing must be kept in mind when defining such objects as lines and points. As it is, an object such as a line, with only one dimension, will only be intersected by chance, and, if one is extraordinarily lucky, will appear as a string of dots. If such objects are required, they should be defined as two- or three-dimensional objects (i.e., a long, thin cylinder for a line or a small sphere for a point) and the anti-aliasing portion of this package improved.

The procedure "pdgetprimitives" must be rewritten to get surfaces defined in the new package. Unused fields (transparency and refractivity in many cases) must be initialized to appropriate values or removed from the ray tracing package completely. If the new package does not specify light sources, either standard light sources should be specified or the capacity to define light sources should be added to the graphics package in question. Major

modifications to the graphics package may have to be made, since this package requires unclipped, three-dimensional objects. The package expects to be able to obtain or calculate surfaces in a left-handed coordinate system where the center of projection is at (0, 0, 0) for a perspective projection, or the front clipping plane is on the x-y plane for a parallel projection. It does not desire the coordinates to be scaled or sheared, however. The window and viewport coordinates must also be available.

The procedure "pddrawrec" must be re-written to either directly call device drivers available to the graphics package or to call routines of the graphics package to draw filled rectangles on the output devices. If the former is chosen, then the procedure must know which device is in use, if multiple devices are available. The maximum screen resolution among the devices for which the ray tracing is to be done must also be known, and if it is greater in either the x- or the y-direction than the current array size to hold previously calculated values in a ray traced picture, then the array size must be changed.

Appendix C: How to Produce a Successful Picture

The following information is intended to be a useful guide to successful picture production using the ray tracing package that has been added to UPCORE as part of this effort. A knowledge of CORE (21) and of the new user functions that have been added to UPCORE (Appendix A) is assumed. No highly theoretical knowledge is assumed in the area of reflectivity, transparency, and refractivity. Such knowledge is useful, however. The attempt is made to give a general discussion of some of the aspects of picture generation and places where a user may encounter difficulty.

Light Sources

The one thing that most pictures need is a light source. While this implementation will operate without a light source; the ray traced output of a scene with no light sources will invariably be a black screen. Not only that, the black screen will generally take a long time to produce.

Any or all of the three kinds of light sources, parallel, point, or ambient, may be used for the purpose of lighting a scene. Each kind of light source has different characteristics, and can be used and combined for different effects.

With all three light sources, it is necessary to make sure that the desired color is set, for the color is as

important as the amount. For example, if the color of a light source is set to black, then the amount is not important since no light will result (black is the absence of light). Also, the amount of each color in the light color will affect the final intensity of the light. Of course, the amount of light must be set to a positive value, or it is the same as no light at all.

Parallel Light Sources. Parallel light sources are the easiest to work with, because there is no distance attenuation of their light. The necessary information to define one is the vector in the direction that the light is to travel in and the intensity and color of its light. For someone using this package for the first time, it is recommended that a large light amount be set, an amount of 20.0 should be sufficient, and that several light sources be placed in such a way as to flood the scene with light. An example would be to create six light sources, one shining along each positive and negative coordinate axis. This has the effect of flooding the scene with light, which helps to determine that the scene looks as was expected. Once the scene arrangement is confirmed, the light sources can then be adjusted, knowing that any strange results are due to the light sources and not to the scene layout.

While parallel light sources are especially nice for testing scenes, they have a few limitations. One limitation is that parallel light sources act as though they were point sources at infinity. Because of this, they cannot be used

to light a totally enclosed area. So a scene of an inside of a house, for example, with no windows or transparent sides, cannot be lit by a parallel light source.

Point Light Sources. In a case where parallel sources are unsuitable, point light sources can be used, and generally present a more natural appearance under the circumstances. Point light sources are trickier to use, since the intensity of their light decreases with the square of the distance from them. This can result in the light decreasing to the point of undetectability before it hits a surface, causing the same results as no light source at all. So when testing point light sources, it is best to start with a very high amount of light, and experiment to find the correct amount.

Ambient Light Source. The third kind of light source, ambient light is a generally useful kind of light, but not well suited as a primary light source. Ambient light radiates from every direction equally, which means that a scene lit by ambient light only will have no shadows, and every surface will be equally lit. Ambient light is generally used to make a scene more natural appearing.

Because ambient light radiates from all directions, it simulates the background light that is caused by low intensity reflections from many surfaces in nature. Natural background light is what causes real shadows to appear more gray than black, because that which is in the shadow is receiving a good deal of indirect light. Ambient light is

usually used for the purpose of preventing shadows from appearing uniformly black.

Ambient light can also be used to set the background color of a scene. The background color that is set in CORE has no effect on the ray tracing package, so to have a background other than black, it is necessary to either place a large surface of another color behind the scene or by setting the ambient light to another color. Putting a large surface behind the scene can cause problems because it may block a light source, and it may not get sufficient lighting to appear the desired color. Also, the shadows from objects in the scene may fall upon such a surface, and show up in the final scene. In addition, it would still be necessary to have ambient light to get non-black shadows.

Surfaces

In addition to light sources, most scenes need surfaces. Surfaces can be defined in CORE as polygons to produce an acceptable picture, but there are now several new attributes that can be used to enhance the picture.

It should be noted that when using UPCORE, the only primitive in UPCORE, not including the light sources that have been added to CORE, that can appear in or affect a ray traced scene are polygons. Text, markers, and lines will currently not appear. The fill color of the polygon is what determines the color of the surface for purposes of ray tracing, and the edge color is disregarded.

Although CORE does not show light sources on the screen, it does show polygons. The first step in any picture generation should be to see the polygons in the scene by using the standard display capabilities present in UPCORE. Although no hidden surfaces will be removed, and the UPCORE polygon fill does not always give a correct fill, it will give some idea of what will appear on the screen when it is ray traced. Warning: the ray tracing package does not support segment transformations. It uses whatever coordinates were defined, plus any transformations used and whatever world coordinate transformation was defined by the user. Aside from that, whatever objects that appear on the screen will appear (unless hidden by other objects) after ray tracing.

Since ray tracing does extensive real number calculations, some glitches can result from round off error. A very common error is where two surfaces are defined so that they overlap on the same plane or have an edge in common. What may happen is that the algorithm will determine that it is intersecting one surface first, and then that the second surface is blocking the light from the first, resulting in a random pattern of black and correctly colored dots over the affected area. This can be remedied effectively by simply defining the surfaces so that they do not superimpose exactly, but rather have a very small distance between them.

There are several new attributes that a user can use to

enhance surfaces definable with UPCORE. It is not necessary to set any of them, since the defaults for these attributes will result in an acceptable surface type.

Reflectivity. One of the most useful new attributes for a simple scene is the reflectivity attribute. The default reflectivity is a diffuse component of 1.0 and a specular component of 0.0. This means that the surfaces are not absorbing any light of their color, a phenomenon which does not occur naturally. One example of this unnaturalness is that a shadow on a white surface will appear the same color as the ambient light. This results in the shadow blending into the ambient light with no visible edge, which is unnatural as well as confusing in appearance. An easy way to produce a more natural appearance without introducing specular reflections is to simply set the reflectivity of a surface to a diffuse component of 0.9 and a specular component of 0.0. This simulates the loss of light due to absorption by the surfaces. Experimentation can also be done to get a particular effect from absorption.

A more complex scene may want to include specular reflections. Specular reflections result in surfaces appearing mirror-like, and can be specified by setting a non-zero specular component of reflectivity. A perfect reflector can be made by setting the specular component of reflectivity to 1.0 and the diffuse reflectivity to 0.0. This will result in a perfectly reflecting surface, regardless of what color the surface was set to, because

specular reflection does not (for purposes of this implementation) depend upon the color of the surface.

For a more natural specular reflection, the specular component should be somewhat less than 1.0, to account for that light that is absorbed by the surface. For the color of the surface to be apparent, the diffuse component of reflection should be greater than 0.0. However, it is also a requirement that the specular component and diffuse component cannot add up to more than 1.0. A diffuse component of 0.4 and a specular component of 0.5 will produce a distinctly shiny surface with a definite color, and with some absorption of light. Experimentation can be done to get other effects desired.

Transparency and Refractivity. More advanced attributes are the related attributes of transparency and refractivity. These attributes can be tricky, and should be handled with care. A good understanding of refractivity is necessary to receive good results.

To begin with, the concept of transparency and refractivity requires the concept of a solid body rather than just the concept of a surface. Since CORE has no provisions to handle solids, it is left to the user to ensure that a transparent object is bounded (closed) by polygonal surfaces. Internally, solids are determined by the surfaces crossed: if a surface has the same refractivity, transparency, and color as the surface that was crossed to enter the current solid, then they are of the

same solid, otherwise they are assumed to be of different objects, and treated accordingly. Since nested solids are acceptable, unusual results can be caused by a inadvertently omitted surface.

Another confusing point is that with this implementation, a surface with a transparency of zero is equivalent to an surface with an infinite refractivity, regardless of what was set. The infinite refractivity means that no light which strikes the surface is refracted through it, so all light is reflected or absorbed, as determined by the specular and diffuse coefficients of reflectivity. This was done to prevent users from needing to know the details of transparency and refractivity. However, for even a nominal transparency, the set refractivity is used to determine reflections and refractions. What can happen is that a transparent or semi-transparent object is set with a refractivity of 1.0, which is the default refractivity and the refractivity of a vacuum. If that object is not enclosed in another transparent object, then the refractivity on the inside and the outside of the object are the same, resulting in no reflection whatsoever from the surface, including the diffuse reflection which would show the color of the object, which is definitely unnatural. However, some change may be made to light coming through the object, dependent on its color and transparency.

Users more familiar with refractivity may want to note that it is possible to simulate a non-transparent object

with a known refractivity by letting the transparency attribute of the surface be less than 10e-10, but greater than zero without the algorithms calculating refracted light rays, which is a time saving feature.

Smoothness. For a more varied surface appearance, the smoothness attribute can be used to vary the way the specular reflection from a light source is spread across a surface. A smoothness of 0.0 results in no spreading. A higher smoothness will result in more spreading. In most cases, a smoothness of 0.1 should be the maximum for this implementation, because the specular reflections of other objects do not spread, regardless of the smoothness, so that a larger value would cause an unnatural look.

Depth

One thing that absolutely must not be forgotten when trying to produce scenes in which specular reflections of objects (not light sources) and refractions are desired is that the ray tree depth determines the maximum number of reflections and refractions of light rays starting at the eye point. With the default depth of 1, only specular reflections of light sources will be found, no object reflections or refractions. A way of determining a reasonable depth is to figure out the maximum number of reflections off of surfaces and refractions through surfaces (not objects) needed to produce the desired effect. Generally, with reflective surfaces, a depth of 3 to 4 is

sufficient. Refractive surfaces may take more, depending on the number of surfaces to be refracted through. While reflective surfaces do not take too much time to calculate, refractive ones will take much more due to the increasing ray tree width as each ray is split into two components while refracting. It should be noted that if there are no surfaces with a non-zero specular reflection coefficient or transparency, the ray tracing depth does not matter, since no reflected or refracted rays will be generated.

Resolution

Since the ray tracing algorithm is basically a sampling technique where areas of the picture are subdivided only when necessary, it is often possible to lose detail such as the points of polygons and the edges of shadows when the resolution is not high enough. While this is generally acceptable when one is trying out a picture for the first time, it is generally less acceptable when the picture is to be reproduced to hard copy. In these cases, the resolution can be changed to a higher amount to generate the final copy. Of course, this will increase the necessary time to produce the picture.

Sometimes, a better way to eliminate the flaws caused by insufficient resolution is to visually determine a problem area. Then set the area resolution higher, redefine the window around the perimeters of the area, and scale and position the viewport accordingly for that portion of the

picture. Now, if a batch of updates is rerun, only the area within the window and viewport will be redone. If, however, the window and viewport are not proportioned correctly, what is produced inside the smaller viewport will not match that which is on the rest of the display surface.

Color Resolution

The color resolution is the largest amount that two colors can differ by and still be considered to be the same. For most applications, it is desirable for the resolution to be high enough, i.e., a small enough difference, so that there is no discernible color change between adjacent color areas of what is supposed to be a gradually shaded surface. However, some applications may require a lower or higher resolution. In general, where time constraints are not of great importance, it is better not to change the color resolution from its default values, which are sufficiently small to prevent discernible color discontinuities.

Invoking the Ray Tracing Package

To invoke the ray tracing package, the display mode should first be set to "hidden surface." In UPCORE, this can be done with the call:

```
setdmode (4)
```

Notice that this call is not allowed during a batch of updates.

Next, a batching of updates must be begun. In UPCORE, the call is:

beginbupdt

After a batch of updates has begun, no changes will be made to the screen until the batch of updates has ended. Because of this, it is possible to begin the batch of updates before all of the scene has been defined, and to see the remaining scene only at the end of the batch of updates.

An end of the batch of updates is indicated in UPCORE by the call:

endbupdt

If the disply mode is 1, this call will cause the scene to be output in "fast" mode, i.e., line drawings only. If the display mode is 2, this call will cause filled polygons to be output (i.e., "fill" mode), if the device supports them. The display mode of 3 is currently not used in UPCORE, but this mode would correspond to "hidden lines." The display mode of 4 is "hidden durface," and this will result in the scene being ray traced, if the device supports it. Notice, in order to end a batch of updates, it is necessary to begin one first.

Trouble Shooting

This section intends to explain some of the causes of more common mistakes which can be made when trying to define a picture.

1. Ran a batch of updates, but it just redrew the same scene that had been produced by CORE without ray

tracing.

(a) Did not set display mode to remove hidden surfaces.

(b) Specified a device that does not support hidden surface removal.

2. Ray traced the picture, but it resulted in a black screen.

   (a) Did not specify light sources properly (black color or zero amount).

   (b) Nothing in window for light to shine on.

3. Ray traced the picture, but it resulted in a screen of the color of the ambient light.

   (a) Nothing in window for light to reflect off of.

   (b) All surfaces perfect reflectors of the ambient light color, and no other light sources shining on visible surfaces.

   (c) Ambient light too bright. Ambient light should seldom have an intensity greater than 1.

4. Specified a reflective surface, but there are no reflections off of it.

   (a) Did not specify reflectivity correctly (e.g., specular coefficient of reflectivity equals 0, or close to it).

   (b) Ray trace depth too low.

(c) Scene set up so nothing is in the proper location to give reflections off of it.

5. Specified a transparent surface, but nothing is refracting through it.
   (a) Did not set up combination of refractivity and transparency correctly (e.g., transparency equals 0, refractivity excessively high (2.4 is the refractivity of diamond; a much higher refractivity is not too good for a trnsparent object)).
   (b) Surface not entirely transparent and is too thick for light to get through.
   (c) Ray trace depth too low.
   (d) Nothing in position to refract through object.

6. Some surfaces look strange--speckled with different colors and/or black.
   (a) Specified two surfaces to occupy the same location.

Appendix D: The Operation of the Devices and Device Drivers Used in This Effort

Two output devices were useful exclusively in this effort. They were a Tektronix 4027 terminal and a Raster Technologies Model One/25S graphics device. Information in this appendix is intended to help explain the operation of these devices and of the device drivers written for them in this effort.

Description of the Raster Technologies Model One/25S Graphics Device

The Raster Technologies graphics device has a 512 by 512 screen with 24 bits per pixel of color. It executes instructions extremely fast, and also has an extremely full instruction set. With it, it is possible to display a wide variety of items in a number of ways and to interact with a user without too much difficulty.

Description of the Tektronix 4027 Graphics Device

The Tektronix 4027 graphics terminal, on the other hand, has only 3 bits of color per pixel, for a total of 8 colors which can be on the screen at once. However, it uses look-up tables, so that there are a total of 64 possible colors. The resolution of this device at 640 by 448 is comparable to that of the Raster Technologies device, but

its instruction set is much more limited, and it executes instructions at a much slower speed than the Model One/25S.

Driver Design

The drivers were written in Pascal, rather than in C, due to unfamiliality with the C progrmming languge. Because of this, there may be some difficulties using them with a language other than Pascal under UNIX.

A good deal of the code for the drivers for the Tektronix 4027 was written to simulate numerous shades of color, from pixels of the 8 colors available at once on the terminal. Code to generate patterns of pixels to simulate the shades was available in the Movie/BYU package (6), and was converted to Pascal for the drivers.

These drivers were designed to match those already written for use with the UPCORE package. Because of this, they do not utilize either of the devices very well.

For example, the Tektronix 4027 has a polygon fill with a specifiable distinct edge color, which is a type of primitive that CORE is supposed to be able to display. However, since UPCORE produces this by filling in the polygon using a scan line method and then drawing the polygon edge separately, no driver was written capable of setting a separate edge color.

Likewise, the Model One/25S has numerous options for a great many things, but because they were not necessary to this effort, they were ignored as far as this driver package

is concerned.

Device Initialization

Both the Raster Technologies Model One/25S and the Tektronix 4027 must be initialized to specific states in order to be run with the device drivers written for this effort on the Vax 11/780 under UNIX. The Model One/25S requires several changes from the device defaults, but these changes can be saved in a non-volatile memoray, so seldom need to be made. The Tektronix 4027, on the other hand, only requires one change, but that change must be made every time the device is turned on.

Initializing the Model One/25S Terminal to Run With This Driver Package. Since the Model One/25S device can store initialization parameters in non-volatile memory, it should not be necessary to re-initialize the device often. However, the non-volatile memory has lost information on occasion, and this terminal may also be used with other computers requiring different settings, so this description of the set-up sequence is included.

The exclamation point preceding the commands to the device is the prompt from the device showing that it is in graphics mode.

1. Cold boot the device. The cold boot button is located on the right-hand rear of the control device (not the monitor). This sets the configuration to whatever is stored in the non-volatile memory.

Table D-1

Sample Configurations of the Raster Technologies Model One/25S Graphics Device

PORT	RTS	CTS	STOP	BITS	XIN	XOUT	CTRL	PARITY	BAUD
ALPHASIO	OFF	OFF	2	8	ON	OFF	ON	NONE	9600
MODEMSIO	OFF	OFF	1	8	ON	OFF	OFF	NONE	1200
GRINSIO	OFF	OFF	2	7	OFF	OFF	OFF	NONE	1200
TABLETSIO	OFF	OFF	2	8	OFF	OFF	OFF	NONE	1200
KEYBSIO	OFF	OFF	1	8	ON	OFF	ON	NONE	300
HOSTSIO	OFF	OFF	2	7	OFF	ON	ON	EVEN	9600

IEEE port : mode =off   address= 0000

Host mode is HEXASCII

ROM sequence number is 019

Special Characters:

EntGr	Break	Warm	Kill	BS	ACK	Abort	Debug	XON	XOFF
0005	0010	001B	0040	0008	0007	0015	0018	0011	0013

2. Type the "enter graphics" character at the keyboard. This should be either a <CTRL D> or a <CTRL E>. When the "enter graphics" character is received from the keyboard, the terminal will display an exclamation point prompt on the screen.

3. Type:

!discfg <CR>

This displays the current configuration of the terminal. If the lines corresponding to the hostsio configuration, the host mode and the special character set match that which is in Table D-1, then the terminal is initialized correctly. If no changes to the configurations are needed, skip the remaining instructions and type:

!quit <CR>

4. If the special character set is different from that which is shown, then type:

!spchar 0, 1, 5 <CR>

!spchar 5, 1, 7 <CR>

The first line sets the "enter graphics" character to a <CTRL E>, and the second line sets the "acknowledge" character to a <CTRL G>.

5. If the hostsio confifigurations are incorrect, type:

!syscfg serial hostsio rfs off cts off stop 2 bits 7 parity e baud 9600 xin off xout on ctrl on <CR>

The terminal will ask:

are you sure?

Type:

yes <CR>

The system will perform a warm boot. Enter graphics mode again by typing a <CTRL E>.

6. If the host mode is incorrect, type:

!syscfg host hostsio ascii

The terminal will ask:

are you sure?

Type:

yes <CR>

Enter graphics mode again by typing <CTRL E>.

7. Save configurations in non-volatile memory by typing:

!savcfg <CR>

The teminal will ask:

are you sure?

type:

yes <CR>

This saves the changes in non-volatile memory, so that if a cold boot is performed, the device will boot with the correct parameters.

Meaning of the Control Characters. The following control characters were established to take the place of <CTRL D> and <CTRL F>, which the Vax 11/780 would not transmit.

<CTRL E>. (supersedes <CTRL D>) This is the new "enter graphics" command character. When received from the host computer, the terminal will remain in graphics mode until an exit "graphics" character string (FF) is received from the host computer (not from the key board). When <CTRL E> is received from the local keyboard, it will remain in graphics mode until the command "QUIT" is received from the key board (not from the host computer).

<CTRL G>. (supersedes <CTRL F>) This is the new "acknowledge" character, which the device must receive from the host computer after sending data to the host computer, before it will continue executing commands.

Initializing the Tektronix 4027 Graphics Terminal to Run With This Package. There are few parameters to be set to use the device with this package. The baud rate needs to be set to 9600 or 1200 to log onto the Vax 11/780, and once logged on, the baud rate should be reset to 2400 so that the

input buffer will not overflow while in heavy use.

1. Set baud rate to 9600 (or 1200) baud. Type:

!bau 9600 <CR>

for 9600 baud, or

!bau 1200 <CR>

for 1200 baud. The exclamation point is the command character for the Tektronix 4027, and must be typed by the user to give a command to the terminal.

2. Log on to Vax as for any other terminal at that baud rate.

3. Set baud rate to 2400 baud. Type:

% stty 2400 <CR>

where the percentage sign is the prompt supplied by the Vax 11/780 under UNIX. This sets the baud rate being sent and received from the Vax 11/780 to 2400 baud.

Type:

!bau 2400 <CR>

where the exclamation point is the command character for the terminal.

Model One/25S Raster Technologies User Subroutines

All of the following procedures are written in Pascal, so problems may be encountered if these are called from a program written in another language.

1. alpharas

Purpose: Take Model One/25S graphics device out of graphics mode.

Calling Sequence: alpharas;

Programming Considerations: Takes terminal out of

graphics mode. It is necessary to be in alpha mode if input from the keyboard is desired.

2. charras

Purpose: Put character string on Model One/25S graphics device screen at current point.

Calling Sequence: charras(num, outary);

```
num : integer; - character number range [0, 80]
ary = array [1..80] of char;
outray : ary - character array;
```

Programming Considerations: Calls initras if not already initialized to graphics mode. Draws text in current color.

3. clrras

Purpose: Flood Model One/25S screen with current color.

Calling Sequence: clrras;

Programming Considerations: Will flood entire screen, regardless of windows. Calls initras if not already initialized to graphics mode.

4. cororgras

Purpose: Set offset of coordinate origin of Model One/25S graphics device.

Calling Sequence: cororgras(x, y);

```
x : integer; - x offset range [-32768, 32767]
y : integer; - y offset range [-32768, 32767]
```

Programming Considerations: Will change all other coordinate registers, so should be used with care. Should usually be used in conjunction with scrorgras and windowras to keep graphics screen coherent. Calls initras if not already in graphics mode.

5. crossras

Purpose: Get location from Model One/25S graphics device screen identified by screen crosshairs and a depressed cursor button from "1" to "F".

Calling Sequence: crossras(ix, iy, icnt);

ix : integer; - returns the x coordinate of crosshairs
iy : integer; - returns the y coordinate of crosshairs
icnt : integer; - returns thedepressed button's numeric equivalent
equivalent

Programming Considerations: Gets first button depressed on cursor after this procedure is called. Button "0" on cursor has no effect. The values of ix and iy are values within the screen parameter coordinates.

6. graphras

Purpose: Put Model One/25S graphics device in graphics mode.

Calling Sequence: graphras;

Programming Considerations: Will cause an error condition if the device has already been placed in graphics mode by host computer. However, if the device was placed in mode by current program, this procedure will not send enter graphics character and so avoid producing an error state in the device.

7. initras

Purpose: Do typical initialization of Model One/25S graphics device.

Calling Sequence: initras;

Programming Considerations: Sets the device to 512 mode, sets lower left hand corner of screen to (0, 0) and window accordingly. Turns cursor off. Clears screen to black and then sets the initial color to white.

8. lineras

Purpose: Put line on Model One/25S graphics device starting at current point in current color.

Calling Sequence: lineras(x, y)

x : integer; - x coordinate of end point of line range [-32768, 32767]
y : integer; - y coordinate of end point of line range [-32768, 32767]

Programming Considerations: Calls initras if not initialized to graphics mode. Only portion of line within current window and screen limits will appear on screen. Initras initializes the window and screen coordinates to [0, 511] by [0, 511]. Endpoint is absolute, not relative.

9. moddisras

Purpose: Set mode of Model One/25S graphics device to 512 mode or 1K mode.

Calling Sequence: moddisras(mode);

```
mode : integer; - 0 for 512 mode
                  1 for 1K mode
```

Programming Considerations: If invalid mode is received, 512 mode is used. Changes most registers. Clears screen.

10. moderas

Purpose: Set way in which new graphics information is added to display.

Calling Sequence: moderas(mode);

```
mode : integer; - 0 for insert data
                  1 for subtract image from data
                  2 for subtract data from image
                  3 for add data to image
                  4 for XOR data to image
                  5 for OR data to image
                  6 for AND data to image
                  7 for write all ones
                  8 for inhibit writing of black
                    pixels
```

Programming Considerations: Mode 0 is system default. Ranges are checked. Negative values are set to 0, values greater than 8 are set to 8.

11. moveras

Purpose: Move current position of Model One/25S graphics beam.

Calling Sequence: moveras(x, y);

```
x : integer; - x coordinate of new position
    range [-32768, 32767]
```

y : integer; - y coordinate of new position
range [-32768, 32767]

Programming Considerations: Ranges are checked. The procedure initras is called if not in initialized state.

12. outras

Purpose: Send hexascii equivalent of a number to Model One/25S graphics devicee.

Calling Sequence: outras(a, places);

a : integer; - number to be sent
range [-32768, 32767]
places : integer; - number of hexadeciamal digits
of output range [1, 4]

Programming Considerations: Should be used carefully, if at all, by a user of this driver package. Will find 2's complement of negative numbers. Ranges are not checked.

13. pointras

Purpose: Put a point on the Model One/25S graphics device at current position in current color.

Calling Sequence: pointras;

Programming Considerations: Calls initras if not already initialized.

14. polyras

Purpose: Put polygon on the Model One/25S graphics device screen relative to the current point in current color.

Calling Sequence: polyras(xarray, yarray, n);
xarray : verarray; - array of x coordinates
range [-32768, 32767]
yarray : verarray; - array of y coordinates
range [-32768, 32767]
verarray = array[1..15] of integer;
n : integer; - number of vertices

Programming Considerations: Calls initras if not initialized. All coordinates are relative to current point at the time of the call. Only coordinates within current window and screen coordinates will appear on screen. Polygons will

be filled if primitive filling is set on by "primfillras."

15. primfillras

Purpose: Set primitives to be either filled or unfilled on Model One/25S graphics device.

Calling Sequence: primfillras(switch);

```
switch : integer; - 0 for unfilled
                    1 for filled
```

Programming Considerations: If invalid switch, set to unfilled. If not initialized, procedure initras is called.

16. recras

Purpose: Put rectangle on Model One/25S graphics device with one corner at current point in current color.

Calling Sequence: recras(x, y);

```
x : integer; - x coordinate of opposite corner
     range [-32768, 32767]
y : integer; - y coordinate of opposite corner
     range [-32768, 32767]
```

Programming Considerations: Ranges checked. If not initialized, initras called. Opposite corner is absolute coordinates, not relative. Only coordinates with in screen coordinates and current window will appear on screen.

17. scolorras

Purpose: Set current color on Model One/25S graphics device

Calling Sequence: scolorras(r, g, b);

```
r : real; - red component of color range [0, 1]
g : real; - green component of color range [0, 1]
b : real; - blue component of color range [0, 1]
```

Programming Considerations: The procedure initras is called if not initialized. Ranges checked.

18. scrorgras

Purpose: Set the coordinates of the point at the center

of screen on Model One/25S graphics device.

Calling Sequence: scrorgras(x, y);

```
x : integer; - x coordinate of center of screen
     range [-32768, 32767]
y : integer; - y coordinate of center of screen
     range [-32768, 32767]
```

Programming Considerations: Best used in conjunction with cororgras and windowras to avoid strange wrap-around. Ranges checked.

19. setcurras

Purpose: Turn alphanumeric cursor on or off on Model One/25S graphics device screen.

Calling Sequence: setcurras(mode);

```
mode : integer; - 0 for off
                  1 for on
```

Programming Considerations: If invalid mode, 0 assumed.

20. windowras

Purpose: Set clipping window on Model One/25S graphics device.

Calling Sequence: window(x1, y1, x2, y2);

```
x1 : integer; - x coordinate of one corner of
     window
y1 : integer; - y coordinate of one corner of
     window
x2 : integer; - x coordinate of other corner of
     window
y2 : integer; - y coordinate of other corner of
     window
     range [-32768, 32767]
```

Programming Considerations: Ranges checked. Parameters rearranged if necessary to conform to device requirement that x1 < x2 and y1 < y2 .

<u>Tektronics 4027 User Subroutines</u>

All of the following procedures are written in Pascal, so problems may be encountered if these are called from a program written in another language.

1. alpha27

Purpose: Enter Tektronics 4027 into alphanumeric mode.

Calling Sequence: alpha27;

Programming Considerations: Clears screen to black.

2. char27

Purpose: Put a line of text on the graphics screen at the current point.

Calling Sequence: char27(num, outray);

```
num : integer; - number of characters to be
printed
     ary = array [1..80] of char;
outray : ary; - character array
```

Programming Considerations: Seems to be some problems drawing through characters on screen. Text color is always white.

3. clr27

Purpose: Clear Tektronics 4027 screen to current color or pattern.

Calling Sequence: clr27;

Programming Considerations: When clearing to a pattern, large areas of screen may change if pattern number is subsequently redefined by this driver package.

4. cross27

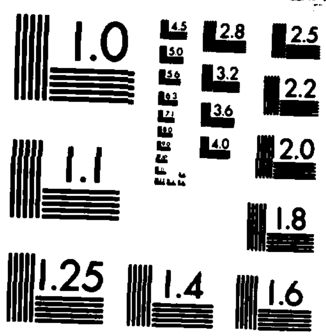
Purpose: Get crosshair location and button pushed.

Calling Sequence: cross27(x, y, icnt);

```
x : integer; - returns x coordinate of crosshair
y : integer; - returns y coordinate of crosshair
icnt : integer; - returns ascii code of key
pressed
```

Programming Considerations: Any non-directional key will return crosshair value. If user is in the habit of randomly pressing keys when program is running, it will cause erroneous results and may cause the program to terminate in an error state.

AD-A153 323 THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO 3/3
A GENERAL PURPOSE..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L R SUZUKI
UNCLASSIFIED DEC 84 AFIT/GCS/MATH/84D-6 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

5. graph27

Purpose: Set up 33 line graphics space and one line work space.

Calling Sequence: graph27;

Programming Considerations: Clears screen to black. Leaves only one line for non-graphics output to appear on.

6. init27

Purpose: Initialize values needed to simulate a wide variety of colors, as well as setting up graphics screen.

Calling Sequence: init27(order);

order : integer; - value 3, 4, 5, 8, 9, or 65 where number of simulated shades is order cubed

Programming Considerations: If an invalid order is sent, order will be set to 65. Clears screen to black. Sets initial color to white. Rearranges color set on terminal from default set.

7. line27

Purpose: Put line on Tektronics 4027 screen.

Calling Sequence: line27(x1, y1, x2, y2);

x1 : integer; - x coordinate of starting point range [0, 639]
y1 : integer; - y coordinate of starting point range [0, 447]
x2 : integer; - x coordinate of ending point range [0, 639]
y2 : integer; - y coordinate of ending point range [0, 447]

Programming Considerations: Does not check range. Start and end points are absolute screen coordinates.

8. move27

Purpose: Move current position of graphics beam.

Calling Sequence: move27(x, y);

```
x : integer; - x coordinate of new position
    range [0, 639]
y : integer: - y coordinate of new position
    range [0, 447]
```

Programming Considerations: Ranges not checked.

9. poly27
   Purpose: Put polygon on Tektronics 4027 screen in current color.

   Calling Sequence: poly27(xarray, yarray, n);

```
xarray : vertarray; - array of x coordinates
     range [0, 639]
yarray : vertarray; - array of y coordinates
     range [0, 447]
     vertarray : array [1..15] of integer;
n : integer; - number of vertices
```

   Programming Considerations: Ranges not checked. Large filled areas may change if current pattern is redefined by this driver package. Coordinates are absolute.

10. rec27

    Purpose: Put rectangle on Tektronics 4027 screen in current color.

    Calling Sequence: rec27(x1, y1, x2, y2);

```
x1 : integer· - x coordinate of first corner
     range [0, 639]
y1 : integer; - y coordinate of first corner
     range [0, 447]
x2 : integer; - x coordinate of opposite corner
     range [0, 639]
y2 : integer; - y coordinate of opposite corner
     range [0, 447]
```

    Programming Considerations: Ranges are checked. Color is "fixed" by breaking up large areas, so pattern numbers can be redefined. Coordinates are absolute.

11. scolor27

    Purpose: Set current color.

    Calling Sequence: scolor27(r, g, b);

```
r : real; - red component of color range [0, 1]
```

g : real; - green component of color range [0, 1]
b : real; - blue component of color range [0, 1]

Programming Considerations: Patterns created to simulate more than the eight possible colors often show distinctly, and large areas may change color if pattern number redefined by this driver package.

12. Internal Procedures:
    - initct27
    - initmpat27
    - map27
    - mpat27
    - scol27
    - spat27

Purpose: Intended to do internal utility processes mainly in support of the color simulation function of this package. Should not be used by users of this driver package.

Appendix E: Statistics on Test Photographs

This appendix lists the attributes of the objects and light sources used to produce the graphical displays, the photographs of which appear in chapter IV. Table E-1, which appears at the end of this appendix, is a table of the user, system and elapsed times needed to produce each display. Some of the times may be high, due to an implementation bug which was discovered too late to regenerate the pictures, but which did not affect the appearance of the pictures, only the times needed to produce them.

Scene Attributes

The important attributes of the objects that appear in the photographs in Chapter IV are listed here. For figures that consist of more than one photograph, the values which vary between the photographs are listed first by photograph, followed by those values common to the entire scene. Photographs generated with UPCORE are so indicated. All others were created with a test bed program, which is capable of defining spheres. Actual sizes and locations are indicated only when pertinent.

```
Figure IV-1
     (a)  Red sphere
               reflectivity: diffuse = 0.8
                             specular = 0.0
     (b)  Red sphere
               reflectivity: diffuse = 0.5
                             specular = 0.3
     (c)  Red sphere
               reflectivity: diffuse = 0.3
                             specular = 0.5
     (d)  Red sphere
               reflectivity: diffuse = 0.0
                             specular = 0.8
     Area resolution: x resolution = 40
                      y resolution = 40
     Color resolution = 0.01
     Ray tree depth = 4
     Ambient light: amount = 1.0
                    color: red = 0.2
                           green = 0.2
                           blue = 0.7
     Light source
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          amount = 2.5
```

```
        direction = (1.0, -1.0, 1.0)
    White plane
        color: red = 1.0
               green = 1.0
               blue = 1.0
        reflectivity: diffuse = 0.8
                      specular = 0.0
        transparency: 0.0
    Red sphere
        color: red = 1.0
               green = 0.0
               blue = 0.0
        transparency = 0.0
        smoothness = 0.01

Figure IV-2
    (a)  Clear sphere
              refractivity = 2.4
    (b)  Clear sphere
              refractivity = 1.5
    (c)  Clear sphere
              refractivity = 1.33
    (d)  Clear sphere
              refractivity = 1.0
    Area resolution: x resolution = 30
                     y resolution = 30
    Color resolution = 0.01
    Ray tree depth = 3
    Ambient light: amount = 1.0
                   color: red = 0.2
                          green = 0.8
                          blue = 0.8
    Light source
        color: red = 1.0
               green = 1.0
               blue = 1.0
        amount = 3.5
        direction = (1.0, -1.0, 1.0)
    Clear sphere
        color: red = 1.0
               green = 1.0
               blue = 1.0
        reflectivity: diffuse = 0.0
                      specular = 1.0
        transparency = 1.0
        smoothness = 0.005

Figure IV-3
    Area resolution: x resolution = 30
                     y resolution = 30
    Color resolution = 0.01
    Ray tree depth = 5
    Ambient light: amount = 1.0
```

```
                    color: red = 0.2
                           green = 0.8
                           blue = 0.8
     Light source
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          amount = 3.5
          direction = (1.0, -1.0, 1.0)
     Outer sphere
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 0.0
                        specular = 1.0
          transparency = 1.0
          refractivity = 1.5
          smoothness = 0.005
          radius = 150.0
     Inner sphere
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 0.0
                        specular = 1.0
          transparency = 1.0
          refractivity = 1.5
          smoothness = 0.005
          radius = 149:5

Figure IV-4
     (a)  Transparent sphere
               refractivity = 2.4
     (b)  Transparent sphere
               refractivity = 1.5
     (c)  Transparent sphere
               refractivity = 1.33
     (d)  Transparent sphere
               refractivity = 1.005
     Area resolution: x resolution = 40
                      y resolution = 40
     Color resoution = 0.01
     Ray tree depth = 4
     Distance scale = 100.0
     Ambient light: amount = 1.0
                    color: red = 0.2
                           green = 0.9
                           blue = 0.9
     Light source
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          amount = 3.5
```

```
        direction = (1.0, -1.0, 1.0)
   White plane
        color: red = 1.0
               green = 1.0
               blue = 1.0
        reflectivity: diffuse = 0.6
                      specular = 0.0
        transparency = 0.0
   Transparent sphere
        color: red = 0.97
               green = 0.97
               blue = 1.0
        reflectivity: diffuse = 0.0
                      specular = 1.0
        transparency = 0.9
        smoothness = 0.005
   Yellow sphere
        color: red = 1.0
               green = 0.8
               blue = 0.2
        reflectivity: diffuse = 0.5
                      specular = 0.5
        transparency = 0.0
        smoothness = 0.05
   Red sphere
        color: red = 1.0
               green = 0.3
               blue = 0.2
        reflectivity: diffuse = 0.5
                      specular = 0.5
        transparency = 0.0
        smoothness = 0.001
   Blue sphere
        color: red = 0.3
               green = 0.6
               blue = 1.0
        reflectivity: diffuse = 0.5
                      specular = 0.2
        transparency = 0.0
        smoothness = 0.005
   White plane
        color: red = 1.0
               green = 1.0
               blue = 1.0
        reflectivity: diffuse = 0.6
                      specular = 0.0
        transparency = 0.0

Figure IV-5
   (a)  Blue sphere
             refractivity = 1.5
   (b)  Blue sphere
             refractivity = 3.0
```

```
    (c)  Blue sphere
              refractivity = 5.0
    (d)  Blue sphere
              refractivity = 100.0
    Area resolution: x resolution = 40
                     y resolution = 40
    Color resolution = 0.01
    Ray tree depth = 4
    Ambient light: amount = 1.0
                   color: red = 0.5
                          green = 0.2
                          blue = 0.5
    Light source
         color: red = 1.0
                green = 1.0
                blue = 1.0
         amount = 3.5
         direction = (1.0, -1.0, 1.0)
    White plane
         color: red = 1.0
                green = 1.0
                blue = 1.0
         reflectivity: diffuse = 0.8
                       specular = 0.0
         transparency = 0.0
    Blue sphere
         color: red = 0.0
                green = 0.0
                blue = 1.0
         reflectivity: diffuse = 0.4
                       specular = 0.5
         transparency = 1.0e-13
         smoothness = 0.01

Figure IV-6
    (a)  Inner sphere
              color: red = 1.0
                     green = 1.0
                     blue = 1.0
              refractivity = 2.4
    (b)  Inner sphere
              color: red = 1.0
                     green = 0.5
                     blue = 1.0
              refractivity = 1.5
    Area resolution: x resolution = 30
                     y resolution = 30
    Color resolution = 0.01
    Ray tree depth = 6
    Ambient light: amount = 1.0
                   color: red = 0.2
                          green = 0.8
                          blue = 0.8
```

```
    Light source
         color: red = 1.0
                green = 1.0
                blue = 1.0
         amount = 3.5
         direction = (1.0, -1.0 1.0)
    Outer sphere
         color: red = 1.0
                green = 1.0
                blue = 1.0
         reflectivity: diffuse = 0.0
                       specular = 1.0
         transparency = 1.0
         refractivity = 1.5
         smoothness = 0.005
    Inner sphere
         reflectivity: diffuse = 0.0
                       specular = 1.0
         transparency = 1.0
         smoothness = 0.005
    White plane
         color: red = 1.0
                green = 1.0
                blue = 1.0
         reflectivity: diffuse = 0.6
                       specular = 0.0
         transparency = 0.0

Figure IV-7
    This scene was defined with the UPCORE package.
    Area resolution: x resolution = 25
                     y resolution = 25
    Color resolution = 0.01
    Ray tree depth = 6
    Ambient light: amount = 0.0
    Light source
         color: red = 1.0
                green = 1.0
                blue = 1.0
         amount = 4.5
         direction = (1.0, -1.0, 1.0)
    Inner cube
         color: red = 0.6
                green = 0.6
                blue = 1.0
         reflectivity: diffuse = 0.0
                       specular = 1.0
         transparency = 1.0
         refractivity = 2.5
    Outer cube
         color: red = 0.6
                green = 0.6
                blue = 1.0
```

```
          reflectivity: diffuse = 0.0
                        specular = 1.0
          transparency = 1.0
          refractivity = 1.5
     White plane
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 1.0
                        specular = 0.0
          transparency = 1.0
```

Figure IV-8

```
     (a)  Black sphere
               smoothness = 0.1
     (b)  Black sphere
               smoothness = 0.01
     (c)  Black sphere
               smoothness = 0.001
     (d)  Black sphere
               smoothness = 0.0001
     Area resolution: x resolution = 40
                      y resolution = 40
     Color resolution = 0.01
     Ray tree depth = 4
     Ambient light: amount = 1.0
                    color: red = 0.2
                           green = 0.2
                           blue = 0.7
     Light source
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          amount = 2.5
          direction = (1.0, -1.0, 1.0)
     White plane
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 0.8
                        specular = 0.0
          transparency = 0.0
     Black sphere
          color: red = 0.0
                 green = 0.0
                 blue = 0.0
          reflectivity: diffuse = 0.2
                        specular = 0.5
          transparency = 0.0
```

Figure IV-9

```
     (a)  Distance scale = 7.0
     (b)  Distance scale = 9.7
```

```
    (c)  Distance scale = 30.0
    (d)  Distance scale = 3000.0
    Area resolution: x resolution = 40
                     y resolution = 40
    Color resolution = 0.01
    Ambient light: amount = 0.0
    Light source
         color: red = 1.0
                green = 1.0
                blue = 1.0
         amount = 600.5
         location = (-150.0, 0.0, 400.0)
    White plane
         color: red = 1.0
                green = 1.0
                blue = 1.0
         reflectivity: diffuse = 0.8
                       specular = 0.0
         transparency = 0.0
         normal to positive z-axis, at a distance
              of 600.0 units from the center of
              projection
    White spheres
         color: red = 1.0
                green = 1.0
                blue = 1.0
         reflectivity: diffuse = 1.0
                       specular = 0.0
         transparency = 0.0
         radii = 25.0
         placed on white plane with their centers in a
              circle of radius 256.0 about the center of
              the plane

Figure IV-10
    (a)  Distance scale = 10.0
    (b)  Distance scale = 100.0
    (c)  Distance scale = 1000.0
    (d)  Distance scale = 1000000.0
    Area resolution: x resolution = 30
                     y resolution = 30
    Color resolution = 0.01
    Ray tree depth = 3
    Ambient light: amount = 1.0
                   color: red = 0.2
                          green = 0.8
                          blue = 0.8
    Light source
         color: red = 1.0
                green = 1.0
                blue = 1.0
         amount = 3.5
         direction = (1.0, -1.0, 1.0)
```

```
Transparent sphere
     color: red = 1.0
            green = 1.0
            blue = 1.0
     reflectivity: diffuse = 0.0
                   specular = 1.0
     transparency = 0.9
     refractivity = 1.5
     smoothness = 0.005

Figure IV-11
     (a)  Ray tree depth = 1
     (b)  Ray tree depth = 2
     (c)  Ray tree depth = 3
     (d)  Ray tree depth = 4
     Area resolution: x resolution = 40
                      y resolution = 40
     Color resolution = 0.01
     Ambient light: amount = 1.0
                    color: red = 0.5
                           green = 0.5
                           blue = 0.5
     Light source
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          amount = 1.5
          direction = (1.0, -1.0, 1.0).
     White plane
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 0.9
                        specular = 0.0
          transparency = 0.0
     Red sphere
          color: red = 1.0
                 green = 0.0
                 blue = 0.0
          refractivity: diffuse = 0.5
                        specular = 0.45
          transparency = 1.0e-13
          refractivity = 1000.0
          smoothness = 0.05
     Blue sphere
          color: red = 0.0
                 green = 0.0
                 blue = 1.0
          reflectivity: diffuse = 0.4
                        specular = 0.55
          transparency = 1.0e-11
          refractivity = 100.0
          smoothness = 0.01
```

```
Green sphere
     color: red = 0.0
            green = 1.0
            blue = 0.0
     reflectivity: diffuse = 0.7
                   specular = 0.0
     transparency = 0.0

Figure IV-12
     (a)  Color resolution = 0.01
     (b)  Color resolution = 0.05
     (c)  Color resolution = 0.15
     (d)  Color resolution = 1.0
     Area resolution: x resolution = 20
                      y resolution = 20
     Ambient light: amount = 1.0
                    color: red = 0.2
                           green = 0.2
                           blue = 0.8
     Light source
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          amount = 2.5
          direction = (1.0, -1.0, 1.0)
     White plane
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 0.8
                        specular = 0.0
          transparency = 0.0
     White sphere
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
          reflectivity: diffuse = 0.7
                        specular = 0.0
          transparency = 0.0

Figure IV-13
     This scene was defined with the UPCORE package.
     (a)  Area resolution: x resolution = 15
                           y resolution = 15
     (b)  Area resolution: x resolution = 50
                           y resolution = 50
     Color resolution = 0.01
     Ray tree depth = 4
     Ambient light: amount = 0.9
                    color: red = 0.55
                           green = 0.7
                           blue = 0.7
     Light source
```

```
        color: red = 1.0
               green = 1.0
               blue = 1.0
        amount = 4.5
        direction = (1.0, -1.0, 1.0)
   White plane
        color: red = 1.0
               green = 1.0
               blue = 1.0
        reflectivity: diffuse = 0.55
                      specular = 0.3
        transparency = 0.0
   Shiny cube
        color: red = 1.0
               green = 1.0
               blue = 1.0
        reflectivity: diffuse = 0.05
                      specular = 0.7
        transparency = 0.0
   Star puzzle
        colors: red, green, blue, cyan, magenta, yellow
        reflectivity: diffuse = 0.8
                      specular = 0.0
        transparency = 0.0

Figure IV-14
   Area resolution: x resolution = 40
                    y resolution = 40
   Color resolution = 0.01
   Ambient light: amount = 1.0
                  color: red = 0.3
                         green = 0.3
                         blue = 0.3
   Red light source
        color: red = 1.0
               green = 0.0
               blue = 0.0
        amount = 1.5
        direction = (0.0, -2.0, -1.0)
   Green light source
        color: red = 0.0
               green = 1.0
               blue = 0.0
        amount = 1.5
        direction = (sqrt(3.0)/2.0, -2.0, 0.5)
   Blue light source
        color: red = 0.0
               green = 0.0
               blue = 1.0
        amount = 1.5
        direction = (-sqrt(3.0)/2.0, -2.0, 0.5)
   White plane
        color: red = 1.0
```

```
              blue = 1.0
              green = 1.0
       reflectivity: diffuse = 0.9
                     specular = 0.0
       transparency = 0.0
  White sphere
       color: red = 1.0
              green = 1.0
              blue = 1.0
       reflectivity: diffuse = 0.9
                     specular = 0.0
       transparency = 0.0

Figure IV-15
  (a)  Upper, lefthand, sphere
            color: red = 1.0
                   green = 0.92
                   blue = 0.75
       Middle, lefthand, sphere
            color: red = 0.35
                   green = 0.1
                   blue = 1.0
       Middle, righthand, sphere
            color: red = 1.0
                   green = 0.8
                   blue = 0.5
  (b)  Upper, lefthand, sphere
            color: red = 1.0
                   green = 1.0
                   blue = 1.0
       Slightly larger sphere around upper, lefthand,
                 sphere
            color: red = 1.0
                   green = 0.92
                   blue = 0.75
            reflectivity: diffuse = 0.0
                          specular = 0.8
            transparency = 1.0
            refractivity = 1.0
            smoothness = 0.005
       Middle, lefthand, sphere
            color: red = 1.0
                   green = 1.0
                   blue = 1.0
       Slightly larger sphere around middle, lefthand,
                 sphere
            color: red = 0.35
                   green = 0.1
                   blue = 1.0
            reflectivity: diffuse = 0.0
                          specular = 1.0
            transparency = 1.0
            refractivity = 2.4
```

```
          smoothness = 0.001
     Middle, righthand, sphere
          color: red = 1.0
                 green = 1.0
                 blue = 1.0
     Slightly larger sphere around middle, righthand,
               sphere
          color: red = 1.0
                 green = 0.8
                 blue = 0.5
          reflectivity: diffuse = 0.0
                        specular = 0.8
          transparency = 1.0
          refractivity = 1.0
          smoothness = 0.005
Area resolution: x resolution = 40
                 y resolution = 40
Color resolution = 0.01
Ray tree depth = 7
Ambient light: amount = 1.0
               color: red = 0.4
                      green = 0.8
                      blue = 0.8
Light source
     color: red = 1.0
            green = 1.0
            blue = 1.0
     amount = 3.5
     direction = (1.0, -1.0, 1.0)
White plane
     color: red = 1.0
            green = 1.0
            blue = 1.0
     reflectivity: diffuse = 0.6
                   specular = 0.0
     transparency = 0.0
Upper, lefthand, sphere
     reflectivity: diffuse = 0.2
                  specular = 0.75
     transparency = 0.0
     smoothness = 0.003
Middle, lefthand, sphere
     reflectivity: diffuse = 0.1
                   specular = 0.7
     transparency = 0.0
     smoothness = 0.004
Middle, righthand, sphere
     reflectivity: diffuse = 0.1
                   specular = 0.7
     transparency = 0.0
     smoothness = 0.008
Lower sphere
     color: red = 1.0
```

```
        green = 0.5
        blue = 0.0
reflectivity: diffuse = 0.9
              specular = 0.0
transparency = 0.0
```

Table E-1. Times to Generate Displays and Output File Sizes

figure		user time	system time	elapsed time	size in bytes
Figure IV-1	(a)	456.9	45.4	3:44:02	472480
	(b)	977.5	207.1	15:00:12	743846
	(c)	909.4	183.3	21:24:45	723119
	(d)	286.6	37.1	34:59	243125
Figure IV-2	(a)	290.9	5.3	33:37	200363
	(b)	420.1	20.2	43:40	251815
	(c)	405.0	17.5	42:04	241040
	(d)	61.4	6.3	23:41	47068
Figure IV-3		2592.2	423.6	68:00:04	331362
Figure IV-4	(a)	1731.9	85.4	13:35:44	794866
	(b)	1961.8	90.3	14:48:20	815542
	(c)	1762.0	78.1	13:53:11	800934
	(d)	1319.0	69.5	10:27:06	717908
Figure IV-5	(a)	435.8	14.8	1:46:19	338653
	(b)	488.8	17.9	10:39	432478
	(c)	761.8	43.1	11:44:46	638433
	(d)	555.1	39.8	3:59:14	539132
Figure IV-6	(a)	4716.1	283.8	24:27:27	523604
	(b)	3269.7	321.1	33:56:27	428023
Figure IV-7		9235.9	618.4	42:37:04	206119
Figure IV-8	(a)	547.1	22.7	2:09:49	558845
	(b)	246.1	4.8	46:40	238640
	(c)	163.8	6.2	15:37	160364
	(d)	176.3	2.8	10:19	147381
Figure IV-9	(a)	1123.5	236.6	22:35:19	751974
	(b)	1954.8	276.2	45:29:13	1250053
	(c)	2879.9	355.7	55:51:16	1852655
	(d)	275.0	14.3	32:22	159857
Figure IV-10	(a)	474.7	10.6	1:38:45	283229
	(b)	--------	unavailable	--------	146711
	(c)	--------	unavailable	--------	243249
	(d)	--------	unavailable	--------	251843
Figure IV-11	(a)	940.6	136.9	21:32:19	823419
	(b)	1065.0	152.9	17:19:47	766592
	(c)	1029.4	95.1	23:06:23	768591
	(d)	1020.6	84.6	6:15:04	768591
Figure IV-12	(a)	376.7	32.9	1:56:52	400956
	(b)	153.0	11.6	34:40	160835
	(c)	106.2	7.3	22:16	100877
	(d)	69.6	2.7	22:05	78021
Figure IV-13	(a)	3613.7	227.1	22:18:06	408488
	(b)	3933.5	255.4	23:04:47	469529
Figure IV-14		763.4	67.3	6:49:53	657523
Figure IV-15	(a)	659.0	46.6	59:40	522230
	(b)	3107.7	491.0	34:14:52	571337

Appendix F: Sample Program

The program listing in Figure F-1 demonstrates the capabilities of the new hidden surface removal capabilities (i.e., the ray tracing package) of UPCORE. The program will draw a scene similar to those in Figure IV-13, but with a different area resolution. The user is given the option of seeing the picture on the Tektronix 4027 terminal or the Raster Technologies Model One/25S graphics device.

The command to compile this program is

```
pc -w puzzle.p lib1 lib2 lib3 lib4
```

where "puzzle.p" is the name that the user has given the program file. Note, if another name is used for this program, it should end in a ".p", so that it can be compiled on the system. The three libraries that were created to hold the compiled UPCORE functions are "lib1", "lib2", and "lib3". "lib4" holds the compiled device drivers. Note, all of these libraries must be present in the directory in which the progam is being compiled, or a path of some sort must exist to them. If these libraries are copied from one directory to another, they must be re-randomized with the command

```
ranlib lib
```

where "lib" is whatever library needs to be randomized. Only one "ranlib" should be running at any one time, because occasionally one will abort for some unknown reason when more than one are running simultaneously.

```
(*************************************************************
*                                                           *
*  date:  2 December 84                                     *
*  version:  1.0                                            *
*  name:  puzzle                                            *
*  description:  This program draws a star shaped puzzle    *
*                sitting on a shiney white plane, next to   *
*                a shiney gray cube.  The scene is first    *
*                produced in line drawings and then         *
*                produced with raytracing.                  *
*  operating system:  VAX 11/780 UNIX                       *
*  language:  pascal                                        *
*  author:  2Lt Laura R. C. Suzuki                          *
*                                                           *
*************************************************************)

program puzzle (input,output);

  const
#include 'defconst.h'

  type
#include 'deftype.h'
    pointtype = record
        x : real;
        y : real;
        z : real;
      end;

  var
#include 'extvar.h'

    xarray, yarray, zarray : rarray;
    device : integer;
    a,b,c : pointtype;
    t1, t2, t3, t4 : real;

#include 'userext.h'

  procedure triangle(a,b,c : pointtype);
(*************************************************************
*  outputs one triangle                                     *
*************************************************************)
    begin
```

Figure F-1. Sample Program

```
      xarray[1] := a.x;
      xarray[2] := b.x;
      xarray[3] := c.x;
      yarray[1] := a.y;
      yarray[2] := b.y;
      yarray[3] := c.y;
      zarray[1] := a.z;
      zarray[2] := b.z;
      zarray[3] := c.z;
      polya3(xarray, yarray, zarray, 3);
    end;

  procedure corner(a,b,c : pointtype);
(*************************************************************
*  defines one corner of the star shaped puzzle             *
*************************************************************)

    var
      t1, t2, t3, center : pointtype;
    begin
      t1.x := a.x;
      t1.y := a.y;
      t1.z := b.z;
      t2.x := c.x;
      t2.y := c.y;
      t2.z := b.z;
      t3.x := c.x;
      t3.y := a.y;
      t3.z := a.z;
      center.x := (a.x + t2.x)/2.0;
      center.y := (a.y + t2.y)/2.0;
      center.z := (a.z + t2.z)/2.0;
      sflndx(1);
      triangle(a, t1, center);
      sflndx(2);
      triangle(a, t3, center);
      sflndx(3);
      triangle(b, t1, center);
      sflndx(4);
      triangle(b, t2, center);
      sflndx(5);
      triangle(c, t2, center);
      sflndx(6);
      triangle(c, t3, center);
    end;
```

Figure F-1 continued.

```
  begin
(* initialize UPCORE *)
    initcore;
(* choose device, TK4027 or Raster Technologies Model
   One/25S *)
    device := 0;
    while (device = 0) do
      begin
        writeln('enter device number');
        writeln('3 - TK4027');
        writeln('4 - Raster Technologies Model One/25S');
        flush;
        readln(device);
        if (device = 3) then
          begin
            initvs(3);
(* viewport of TK4027 terminal is not square, so the
   viewport should be reset *)
            svprt2(0.0,1.0,0.0,0.7);
            swindo(-700.0, 2300.0, -1050.0, 1050.0);
          end
        else if (device = 4) then
          begin
            initvs(4);
            swindo(-500.0, 2000.0, -1250.0, 1250.0);
          end
        else
          begin
            device := 0;
            writeln('try again');
          end;
      end;
(* depth of ray tree *)
    setdepth(3);
(* minimal area resolution *)
    setres(40,40);
(* various colors to be used *)
    dclndx(1, 1.0, 0.0, 1.0);  (* magenta *)
    dclndx(2, 1.0, 0.0, 0.0);  (* red *)
    dclndx(3, 1.0, 1.0, 0.0);  (* yellow *)
    dclndx(4, 0.0, 1.0, 0.0);  (* green *)
    dclndx(5, 0.0, 1.0, 1.0);  (* cyan *)
    dclndx(6, 0.0, 0.0, 1.0);  (* blue *)
    dclndx(7, 1.0, 1.0, 1.0);  (* white *)
```

Figure F-1 continued.

```
    dclndx(8, 0.55, 0.7, 0.7);  (* off grey *)
(* view reference point *)
    svrfpt(300.0,100.0,150.0);
(* vector in up direction *)
    svup3(0.0,0.0,1.0);
(* viewplane normal *)
    svpnor(-3.0,-1.0,-1.5);
(* set to perspective projection *)
    sproj(2,12000.0,4000.0,6000.0);
(* front and back clipping planes *)
    svdpth(1.0,10000.0);
(* view plane distance *)
    svpdis(350.0);
(* background to white *)
    sbgndx(7);
(* so start out with white screen *)
    newframe;
(* set up light sources *)
    crrseg('light');
      setltndx(7);
      setltamt(4.5);
(* parallel source *)
      palsabs(-1.5,1.0,-1.25);
(* ambient light *)
      setaml(8,0.9);
    clrseg;
    ssvis('light', true);
(* set up puzzle *)
    crrseg('puzzle');
(* set faces of puzzle to be diffuse reflectors with a
   coefficient of 0.8 *)
      setrfl(0.8, 0.0);
(* define puzzle *)
      a.y := 200.0;
      b.z := 0.0;
      c.x := 0.0;
      a.x := -300.0;
      while a.x <= 300.0 do
        begin
          b.x := a.x;
          a.z := -300.0;
          while a.z <= 500.0 do
            begin
              c.z := a.z;
              b.y := -100.0;
```

Figure F-1 continued

```
                while b.y <= 500.0 do
                  begin
                    c.y := b.y;
                    corner(a, b, c);
                    b.y := b.y + 600.0;
                  end;
                a.z := a.z + 600.0;
              end;
            a.x := a.x + 600.0;
          end;
(* define plane puzzle is sitting on *)
      xarray[1] := -2000.0;
      xarray[2] := -2000.0;
      xarray[3] := 2000.0;
      xarray[4] := 2000.0;
      yarray[1] := -1000.0;
      yarray[2] := 3000.0;
      yarray[3] := 3000.0;
      yarray[4] := -1000.0;
      zarray[1] := -300.001;
      zarray[2] := -300.001;
      zarray[3] := -300.001;
      zarray[4] := -300.001;
(* set plane to be partially diffuse and partially
   specularly reflecting *)
      setrfl(0.55, 0.30);
(* set plane to be white *)
      sflndx(7);
(* define plane *)
      polya3(xarray,yarray,zarray,4);
(* set cube to be mostly specularly reflecting *)
      setrfl(0.05, 0.7);
(* define faces of cube *)
      xarray[1] := -400;
      xarray[2] := 1000;
      xarray[3] := -400;
      xarray[4] := -1800;
      yarray[1] := 500;
      yarray[2] := 1900;
      yarray[3] := 3300;
      yarray[4] := 1900;
      zarray[1] := -300;
      zarray[2] := -300;
      zarray[3] := -300;
      zarray[4] := -300;
```

Figure F-1 continued.

```
polya3(xarray, yarray, zarray, 4);
zarray[1] := 1700;
zarray[2] := 1700;
zarray[3] := 1700;
zarray[4] := 1700;
polya3(xarray, yarray, zarray, 4);
xarray[1] := -400;
xarray[2] := 1000;
xarray[3] := 1000;
xarray[4] := -400;
yarray[1] := 500;
yarray[2] := 1900;
yarray[3] := 1900;
yarray[4] := 500;
zarray[1] := -295;
zarray[2] := -295;
zarray[3] := 1700;
zarray[4] := 1700;
polya3(xarray,yarray,zarray,4);
zarray[1] := -295;
zarray[2] := -295;
zarray[3] := 1700;
zarray[4] := 1700;
xarray[1] := -1800;
xarray[2] := -400;
xarray[3] := -400;
xarray[4] := -1800;
yarray[1] := 1900;
yarray[2] := 3300;
yarray[3] := 3300;
yarray[4] := 1900;
polya3(xarray, yarray, zarray, 4);
xarray[1] := -400;
xarray[2] := -1800;
xarray[3] := -1800;
xarray[4] := -400;
yarray[1] := 500;
yarray[2] := 1900;
yarray[3] := 1900;
yarray[4] := 500;
polya3(xarray, yarray, zarray, 4);
xarray[1] := 1000;
xarray[2] := -400;
xarray[3] := -400;
xarray[4] := 1000;
```

Figure F-1 continued.

```
      yarray[1] := 1900;
      yarray[2] := 3300;
      yarray[3] := 3300;
      yarray[4] := 1900;
      polya3(xarray,yarray,zarray,4);
    clrseg;
    ssvis('puzzle',true);
(* set display mode to remove hidden surfaces *)
    setdmode(4);
(* start batch of updates *)
    beginbupdt;
(* end batch of updates *)
    endbupdt;
(* terminate device surface *)
    termvs;
    flush;
    readln;
(* terminate UPCORE *)
    termcore;
  end.
```

Figure F-1 continued.

Bibliography

1. Appel, Arthur. Some techniques for shading machine renderings of solids. SJCC Washington, D. C.: Thompson Books, 37-45 (1968).

2. Artherton, Peter and others. "Polygon Shadow Generation," SIGRAPH '78 proceedings, published as Computer Graphics, 12(3): 275-281 (Aug. 1978).

3. Blinn, James F. "Simulation of Wrinkled Surfaces," SIGGRAPH '78 Proceedings, published as Computer Graphics, 12(3): 286-292 (Aug. 1978).

4. Blinn, James F. and Martin E. Newell. "Texture and Reflection in Computer Generated Images," Communications of the ACM, 19: 542-547 (Oct. 1976).

5. Born, Max and Emil Wolf. Principles of Optics. (Second Edition) New York: The MacMillan Co., 1964.

6. Christiansen, Hank and Mike Stephenson. Movie.BYU. July 1981.

7. Cook, Robert L. and Kenneth E. Torrance. "A Reflectance Model for Computer Graphics," Computer Graphics, 15: 307-316 (Aug. 1981).

8. Crow, Franklin C. "The Aliasing Problem in Computer-Generated Shaded Images," Communications of the ACM, 20: 799-805 (Nov. 1977).

9. Dungan, William, Jr. and others. "Texture Tile Considerations for Raster Graphics," SIGGRAPH '78 Procedings, published as Computer Graphics, 12(3): 130-134 (Aug. 1978).

10. Foley, J. D. and A. Van Dam. Fundamentals of Inteactive Computer Graphics. Reading, Mass.: Addison-Wesley Publ. Co., 1982.

11. Franklin, Wm. Randolph. "A Linear Time Exact Hidden Surface Algorithm," Computer Graphics, 14: 117-123 (July 1980).

12. Goldstein, Robert A. and Roger Nagel. "3-D Visual simulation," Simulation, 16: 25-31 (Jan. 1971).

13. Hanrahan, Pat. "Ray Tracing Algebraic Surfaces," Computer Graphics, 17: 83-90 (July 1983).

14. Kajiya, James T. "New Techniques for Ray Tracing Procedurally Defined Objects," Computer Graphics, 17: 91-102 (July 1983).

15. -----. "Ray Tracing Parametric Patches," Computer Graphics, 16: 245-254 (July 1982).

16. Kay, Douglas Scott and Donald Greenberg. "Transparency for Computer Synthesized Images," SIGGRAPH '79 Proceedings, published as Computer Graphics, 13(2): 158-164 (Aug. 1979).

17. Lane, Jeffrey M. and others. "Scan Line Methods for Displaying Parametrically Defined Surfaces," Communications of the ACM, 23: 23-34. (Jan 1980).

18. Nussbaum, Allen and Richard A. Phillips. Contemporary Optics For Scientists and Endineers. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1976.

19. Phong, Bui Tuong. "Illumination for Computer Generated Pictures," Communications of the ACM, 18: 311-317 (June 1975).

20. Raster Technologies. Raster Technologies Model One/25 Programming Guide. Revision 4.1, Dec. 12, 1983.

21. "Status Report of the Graphics Standards Planning Committee," Computer Graphics, 13(3) (Aug. 1979).

22. Taylor, John W. Implementation and Evaluation of a CORE Graphics System on a Vax 11/780 With a UNIX Operating System. MS thesis, AFIT/MA/GCS/83D-8. School of Engineering, Air Force Institute of of Technology (AU), Wright-Patterson AFB OH, December 1983.

23. Tektronix, Inc. 4027A Color Graphics Terminal. Programmer's Reference Manual. 1981.

24. Wailes, Tom Wailes. Placing Hidden Surface Removal Within the CORE Graphics Standard: an Example. MS thesis, AFIT/GCS/MA/83D-9. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1983.

25. Whitted, Turner. "An Improved Illumination Model for Shaded Display," Communications of the ACM, 23 343-349 (June 1980).

Vita

Second Lieutenant Laura R. C. Suzuki was born on 6 October 1962 in Auburn, Washington. She graduated from high school in Tunkhannock, Pennsylvania, in 1980, and attended Wilkes College, Wilkes-Barre, Pennsylvania, from which she received a Bachelor of Science in Mathematics and Computer Science in May 1983. Upon graduation, she received a commission in the USAF through the ROTC program. She entered the School of Engineering, Air Force Institute of Technology, in June 1983.

Permanent Address: 732 River Road
Binghamton, N.Y. 13901

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/MATH/84D-5	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) See Box 19	

12. PERSONAL AUTHOR(S)
Laura R. C. Suzuki, B.S., 2d Lt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM ____ TO ____	1984 December	219

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	
09	02		Graphics, Computer Graphics, CORE

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: THE ADDITION OF ADVANCED SCENE RENDERING TECHNIQUES TO A GENERAL PURPOSE GRAPHICS PACKAGE.

Thesis Advisor: Charles W. Richard Jr.
Associate Professor of Mathematics

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Charles W. Richard Jr.	22b. TELEPHONE NUMBER (Include Area Code) 513-255-3098 22c. OFFICE SYMBOL AFIT/ENC

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

SECURITY CLASSIFICATION OF THIS PAGE

A method for generating realistic scenes by computer graphics was investigated. The algorithm which was used was a ray tracing algorithm. The scenes it was capable of rendering included those containing transparent and reflective surfaces. The implemented surface types were planar polygons and spheres. Minor surface irregularities were simulated for specular reflection from light sources. The resulting package was added to an implementation of a CORE graphics system, to serve as its hidden surface removal facility.

SECURITY CLASSIFICATION OF THIS PAGE

END

FILMED

5-85

DTIC